Ico Dânger Vicente

Aplicação de MonetDB na avaliação de desempenho de bases de dados verticais

Universidade Fernando Pessoa – Porto, 2015

Universidade Fernando Pessoa – Porto, 2015

Ico Dânger Vicente

Aplicação de MonetDB na avaliação de desempenho de bases de dados verticais

Universidade Fernando Pessoa – Porto, 2015

Aplicação de MonetDB na avaliação de desempenho de bases de dados verticais

Por:

Ico Dânger Vicente

Orientador

Professor Doutor Feliz Ribeiro Gouveia

Dissertação apresentada à Universidade Fernando Pessoa como parte de requisitos para a obtenção do grau de Mestre em Engenharia Informática – Sistemas de Informação e Multimédia

Resumo

Esta dissertação analisa a aplicação do Sistema de Gestão de Bases de Dados MonetDB na avaliação do desempenho de bases de dados verticais, comparando com os sistemas PostgreSQL e CitusDB.

Nos últimos anos, os sistemas de bases de dados verticais têm atraído muito interesse não só na comunidade científica como também nas comunidades empresarial e organizacional. Esse interesse está relacionado com o potencial de melhor desempenho, com a forma como as bases de dados são armazenadas, com a possibilidade de compressão dos dados e com o seu suporte no apoio à decisão nas organizações. O interesse crescente no uso de bases de dados por colunas em relação às bases de dados tradicionais, com armazenamento por linhas, deve-se essencialmente à forma de armazenamento e ao desempenho. Os sistemas de base de dados por linhas armazenam os registos de uma relação de forma sequencial, por página, enquanto os sistemas de bases de dados em coluna armazenam os valores pertencendo à mesma coluna de forma contínua, na mesma página, o que torna mais rápidas as operações de leitura de apenas um subconjunto das colunas de uma tabela. Nesta dissertação descrevem-se as principais características e vantagens do método de armazenamento por colunas em relação ao método de armazenamento por linhas, analisando sua arquitetura e os conceitos, e analisando as vantagens da compressão e das técnicas de materialização na execução de consultas. Essas vantagens mostram que a nível de execução de consultas típicas de aplicação analíticas, o desempenho das bases de dados por linhas é inferior ao das bases de dados por colunas coluna.

Abstract

This dissertation analyzes the application of MonetDB in a performance evaluation of vertical databases against traditional systems as PostgreSQL and CitusDB.

In recent years, vertical database systems have attracted great interest both in the scientific community as well as in commercial areas. This interest is related to performance issues, to how the databases are stored, to the use of data compression and to their use in decision support queries. The growing interest in the use of vertical, or columnar, databases over traditional database storage lies mainly in the way data storage is made and to performance gains in some situations. The traditional database systems store tuples sequentially, by page, while vertical database systems store data belonging to the same column continuously, in the same page, which makes it faster to read a subset of a table. This dissertation describes the main characteristics and advantages of the vertical storage method in relation to the traditional storage method, analyzing its architecture and concepts, highlighting the compression advantages and materialization in the analysis of queries. These advantages show that the level of query execution performance of traditional databases, for analytical applications, is slower than the vertical databases.

Agradecimentos

Agradeço em primeiro lugar a minha família pela força e confiança depositada em mim mesmo sabendo das minhas necessidades nunca deixaram de acreditar mesmo nos momentos mais défices e nas horas das decisões.

O meu voto de agradecimento também se direciona a todos os professores da Universidade Fernando Pessoa pelo enorme aprendizado, apoio e carinho, principalmente pelo meu orientador e professor Doutor Feliz Ribeiro Gouveia pela colaboração e conselho em assuntos diversos referentes ao tema desta dissertação e disponibilidade na concretização desta dissertação.

Agradeço muito em especial à minha esposa e à minha filha por esse período de ausência da minha companhia.

Índice

Índice de figuras

Índice de tabelas

Lista de Siglas e Acrónimos

BAT	Binary Association Tables
DW	Data Warehouse
DSM	Decomposition Storage Model
EM	Early Materialization
LM	Late Materialization
MAL	Monet Assembly Language
NoSQL	Not only SQL
NSM	N-ary Storage Model
OLAP	On-Line Analytical Processing
OLTP	On Line Transaction Processing
SGBD	Sistema de Gestão de Bases de Dados
SQL	Structured Query Language
SSBM	Star Schema Benchmark
TPC-H	Transaction Processing Council benchmark H
WAL	Write Ahead Logging

1 Introdução

A forma como os registos são armazenados em memória não-volátil (geralmente em discos) influencia o desempenho do SGBD porque o custo de acesso constitui a maior fração do custo de execução de consultas. O SGBD usa o gestor de armazenamento e o gestor de memória tampão para transferir e armazenar dados em disco. Ao longo dos anos, os SGBD têm sofrido grandes evoluções, principalmente ao nível do desempenho e das possibilidades de gestão. O desempenho é essencial em quase todos os sistemas de bases de dados. Sendo o desempenho um fator-chave, o objetivo desta dissertação é o de avaliar o desempenho do SGBD MonetDB, um dos produtos que utiliza uma forma de armazenamento diferente da tradicional, e que apresenta uma adoção crescente.

Existem vários modelos de Base de Dados: o Modelo em Rede, o Modelo Hierárquico, o Modelo Relacional, o Modelo Orientado a Objetos e o Modelo Objeto-Relacional (Gouveia, 2014). O Modelo relacional (Codd, 1970) além de ser o primeiro modelo teoricamente descrito, destaca-se desde a sua conceção devido a ser de fácil definição e manipulação utilizando uma linguagem de consulta, SQL (Structured Query Language). O modelo define regras que garantem a consistência e a integridade dos dados. No modelo relacional os dados são armazenados em estruturas tabulares, também chamadas relações, onde as linhas correspondem às entidades do mundo real e as colunas correspondem aos atributos dessas entidades. Por exemplo, se uma tabela contém dados armazenados por uma empresa acerca dos seus fornecedores, cada linha contém informações sobre os fornecedores e cada coluna contém um atributo específico referente a cada fornecedor, como por exemplo o nome, ou o endereço.

Tradicionalmente os SGBD armazenam em disco as linhas de uma relação em blocos de comprimento fixo, tipicamente 8K bytes. O número de linhas por bloco depende do seu tamanho; é comum blocos sequenciais em disco armazenarem linhas da mesma relação. Um modelo alternativo de armazenamento mantém num mesmo bloco apenas uma coluna de cada uma das linhas (por exemplo, todos os nomes dos fornecedores mo mesmo conjunto de blocos, em seguida todos os endereços noutro conjunto de blocos, e assim sucessivamente). Neste caso o número de linhas por bloco é maior, porque o tamanho de cada linha corresponde apenas ao tamanho de uma das suas colunas. Se

uma consulta precisar de ler mais do que uma coluna, a junção para obter a linha pretendida tem que ser feita em memória.

A escolha de uma ou outra destas abordagens é feita com a base na expectativa de um melhor desempenho. Em torno dos anos 90, surge a necessidade do uso de base de dados para processamento de consultas analíticas. Assim, além do uso bases de dados para automatizar processos de negócios, as empresas começaram a usar bases de dados para ajudar nas tomadas de decisão e planeamento. Novos problemas surgiram nesta nova forma de uso de bases de dados. A execução de consultas analíticas é tipicamente mais demorada porque contempla um volume muito maior de dados, e as transações de pequenas consultas teriam que bloquear até ao fim do processamento das consultas analíticas (para evitar a inconsistências de leitura). As consultas analíticas e as consultas transacionais em geral não processam os mesmos dados, assim, as organizações tendem a criar duas bases de dados, em vez de uma única. As consultas transacionais são executadas numa base de dados transacional e as consultas analíticas são executadas no que se denomina geralmente um armazém de dados, contendo grandes volumes de dados, geralmente históricos.

Devido às características de cada tipo de consultas, cada base de dados deve ser desenhada de forma diferente. As consultas transacionais podem ser de leitura ou de escrita e tipicamente acedem um grande conjunto de dados de poucas tabelas, enquanto as consultas analíticas acedem a muitas tabelas, algumas das quais com uma grande quantidade de dados. As junções necessárias para responder a uma consulta analítica demorariam muito tempo a responder numa base de dados transacional. Acresce que as consultas analíticas são apenas de leitura, o que permite otimizar a gestão das transações, nomeadamente no que diz respeito ao mecanismo de controlo de concorrência a utilizar.

Uma das alternativas para contornar os problemas inerentes às bases de dados relacionais para processamento de consultas analíticas é o uso de bases de dados geralmente agrupadas sob a denominação NoSQL (Not only SQL). As bases de dados NoSQL têm sido usadas com sucesso para processar grandes volumes de dados, sendo aplicáveis em situações com características próprias tal como o processamento de consultas em bases de dados com grandes volumes de dados e de inserções de novos dados, como é o caso das redes sociais, sistemas de mensagens, e portais de leilões.

O modelo colunar vem sendo amplamente aplicado na substituição da metodologia de armazenamento em linhas, devido a ambos tratarem do mesmo tipo de dados, trabalharem bem com SQL e pela sua superioridade em desempenho com grandes quantidades de dados para certas aplicações.

Nos últimos anos houve introdução de uma serie de sistemas de bases de dados orientadas por coluna, incluindo MonetDB. Esses sistemas têm atraído muita atenção devido à forma com que os dados são armazenados e recuperados. Muitas das empresas que trabalham com grandes volumes de dados, como Google e Facebook, em que a evolução dos dados cresce de forma exponencial, adotaram os modelos de bases de dados orientados por coluna (Cattel, 2010).

1.1 Descrição do problema

Nos últimos anos o volume de dados gerados tem vindo a aumentar exponencialmente, trazendo desafios tecnológicos aos sistemas que gerem o seu armazenamento, acesso, pesquisa e recuperação em caso de falha.

A maioria dos SGBD relacionais tem por armazenamento linha por linha. Este tipo de armazenamento apresenta desvantagem quando é necessário ler somente pequenas colunas de todas as linhas. Pelo facto de dados ser armazenado linha por linha, uma simples consulta de dados, leva-nos a percorrer todas as linhas.

O problema recai quando esses dados são armazenados em grandes páginas, de modo que, grandes números de itens de dados podem ser recuperados em uma única operação de leitura com finalidade a tomada de decisões, resultando em uma taxa de acerto em geral elevada.

Se considerarmos a situação de armazenamento de grandes volumes de dados sob múltiplas perspetivas, tomadas das decisões em tempo real, consultar dados específicos (que se encontram em uma coluna) desta tabela e retornar somente dados referentes à coluna solicitada, são tarefas que processam muitas linhas e poucas colunas.

O armazenamento linha por linha para este tipo de situação, não será vantajoso, em que queremos selecionar todos os valores de uma tabela referente a uma linha específica. O

resultado desta seleção muitas vezes seleciona pequenas linhas mas possivelmente todas as colunas.

Neste contexto, o desenvolvimento deste trabalho pretende analisar o desempenho de grandes volumes de bases de dados verticais usando o SGBD MonetDB5 e representantes de SGBD com armazenamento tradicional.

1.2 Objetivos do trabalho

O trabalho a desenvolver tem por objetivo usar MonetDB5 para avaliar o desempenho em consultas complexas com interesse na avaliação dos sistemas de bases de dados orientados por coluna num sistema de armazém de dados, por exemplo, combinar tabelas com centenas de colunas e vários milhões de linhas, em tempo real, que não será possível utilizar a tecnologia tradicional de banco de dados.

O crescimento explosivo de armazenamento de dados, apoio à decisão e aplicações de análise de dados gerou uma urgência para ler e processar conjuntos de dados muito grandes de forma rápida e precisa em tempo útil. Este crescimento de volumes de dados e expectativas crescentes são um desafio para manter ou melhorar através de melhorias de desempenho incrementais.

1.3 Motivação

Sendo o tempo de resposta um fator cada vez mais importante no meio empresarial, e um dos principais impulsionadores para o crescimento e eficácia nas organizações, o desempenho dos sistemas de armazém de dados é crucial para determinar o seu sucesso e, consequentemente, o das organizações associadas.

Uma das razões que motivou a realizar este trabalho foi a abordagem do de bases de dados verticais numa das disciplinas no decorrer do curso, que motivou-me a querer obter mais informações sobre esta nova forma de armazenamento.

O segundo motivo é por ser um tema basicamente novo na área de base de dados e, sendo assim, existem poucos trabalhos sobre este tema que descrevem os sistemas de base de dados orientados a coluna com resultados que normalmente incluem ganhos de desempenho em relação a bases de dados relacionais.

1.4 Metodologia

A metodologia usada na descrição deste trabalho baseia-se na implementação e execução de testes, utilizando padrão reconhecidos, como o TPC-H (TPC, 2014), em particular:

- Analisar as características da base de dados MonetDB5 através da base de dados TCP-H.
- Analisar os resultados e obter conclusões acerca do desempenho do MonetDB5, PostgreSQL 9.4 e CitusDB, para um nível limitado de carga de trabalho.

Tanto quanto possível foram mantidas condições experimentais iguais para os três produtos, e foram recolhidos resultados médios de seis execuções sequenciais dos testes para cada um. Todos os produtos foram utilizados com as suas configurações por omissão, correspondendo às sugestões das equipas que os desenvolvem. Optou-se por não modificar as configurações para não favorecer uns produtos em detrimento de outros, uma vez que as alterações são muito específicas e implicariam um grande conhecimento do comportamento dos três produtos, conhecimento que não seria possível durante o curto período de realização deste trabalho.

1.5 Estrutura do documento

Esta dissertação está organizada em capítulos, sendo este primeiro capítulo uma introdução sobre o tema a ser abordado, sobre os objetivos do trabalho e a metodologia adotada. Os restantes capítulos são:

Capítulo 2, descreve os Sistemas de Bases de dados relacionais. Este capítulo apresenta uma breve introdução sobre bases de dados relacionais, abordando questões diversas acerca dos sistemas de bases de dados relacionais e sistemas de bases de dados por colunas, tais como os tipos mais comuns de armazenamento em disco.

Capítulo 3, descreve as bases de dados alternativas ao modelo relacional, NoSQL e orientadas por colunas. São apresentadas as características e a arquitetura das bases de dados por colunas, bem como as vantagens e desvantagens entre os sistemas de bases de dados por colunas e bases de dados relacionais, abordando-se em particular técnicas para melhorar a desempenho dos sistemas de bases de dados orientadas por colunas.

Capítulo 4, apresenta o problema, começando com uma apresentação de armazém de dados. Descrevem-se os principais pontos que devem ter em conta na sua implementação, bem como o esquema estrela que é usado nesta dissertação. Apresenta-se também a estrutura e a arquitetura do MonetDB, e faz-se uma breve introdução aos outros SGBD, PostgreSQL e CitusDB.

Capítulo 5, descreve os testes, e são apresentados os meios usados na fase experimental de forma a criar os suportes necessários na comparação dos sistemas de base de dados relacionais e sistemas de base de dados em colunas. É feita também uma análise dos resultados obtidos.

Capítulo 6, apresenta as conclusões do estudo descrito nesta dissertação, bem como o trabalho futuro que poderá ser desenvolvido de forma a melhorar a avaliação do desempenho deste tipo de produtos.

O anexo A apresenta a listagem as consultas utilizadas nos testes.

2 O Modelo Relacional

2.1 Introdução

Na década de 70 e 80 deram-se os primeiros passos na implementação da base de dados, e aplicações pioneiras utilizavam estrutura de modelo hierárquicos e modelos em rede. Estes tipos de estruturas não apresentavam flexibilidade e a probabilidade dos registos serem duplicados era enorme, apesar da eficiência de transações e no acesso a consulta. Com o desenvolvimento no campo da investigação e as necessidades do mercado, houve a necessidade de melhorias e simplificação da iteração com o sistema, nomeadamente das consultas. Os sistemas de gestão de bases de dados relacionais surgiram para colmatar essas necessidades e para permitir o processamento de consultas arbitrárias, aumentando a produtividade dos programadores (Codd, 1970). O SGBD é responsável por gerir concorrência, manter o isolamento entre transações de diferentes aplicações, e permitir recuperação em caso de falha, garantindo que as transações são atómicas e duráveis. A arquitetura clássica cliente-servidor é mostrada na figura seguinte:

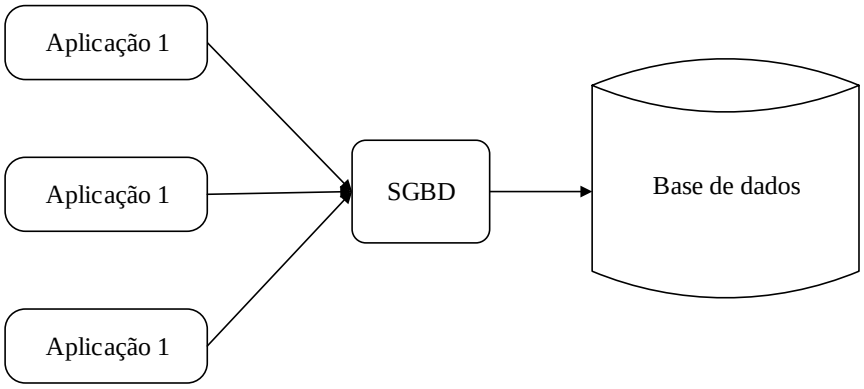


Figura 2-1 Sistema de Gestão de bases de dados.

Este sistema apresenta a informação de forma mais compacta o que permite o acesso simultâneo e o compartilhamento dos dados, evitando a duplicação da informação e permitindo uma recuperação e modificação mais rápida dos dados. É um modelo baseado em lógica de predicados e na teoria de conjuntos, utilizando conceitos de entidade e de relação.

Uma entidade é um elemento caracterizado pelos dados que são recolhidos na sua identificação, e é vulgarmente designada por tabela, ou relação. Um relacionamento determina o modo como cada registo de cada tabela se associa a registos de outras tabelas. Na prática, entidades e relacionamentos são representados no modelo relacional por tabelas.

O modelo relacional também introduziu uma linguagem de consulta de alto nível, SQL, melhorando o desempenho das consultas. Para trabalhar com tabelas relacionais, algumas restrições foram impostas para evitar efeitos indesejáveis, tais como:

- Repetição de informação, introduzindo redundância, e originando os problemas de consistência nas operações de atualização, de inserção e de remoção de informação.

- Incapacidade de representar partes da informação, garantindo que não há informação ambígua ou inconsistente.

- Perda de informação, significando que a junção de duas relações pode dar mais informação que a contida na relação original que lhes deu origem. Desta forma garante-se que um esquema de bases de dados pode ser normalizado.

Essas restrições são conhecidas como restrições de integridade referencial, restrições de chaves e restrições de integridade de junções. A teoria associada que permite evitar estes problemas chama-se normalização, e diz-se que as relações respeitam uma dada forma normal. As formas normais mais comuns são a terceira forma normal, e a forma normal de Boyce-Codd. O processo de normalização garante que o esquema da base de dados evita um determinado conjunto de anomalias, embora, ao aumentar o número de relações ligadas entre si, possa influenciar negativamente o desempenho de consultas ao obrigar a mais junções. A introdução à arquitetura clássica do SGBD e ao Modelo Relacional pode ser encontrada em qualquer livro sobre o tema, por exemplo (Gouveia, 2014)(Hellerstein, Stonebraker, *et al*., 2007).

A figura seguinte mostra o conjunto de tabelas para representar as relações Aluno, Disciplina e Inscrito. As setas na figura representam relações de integridade referencial, ligando as chaves estrangeiras às respetivas chaves primárias.

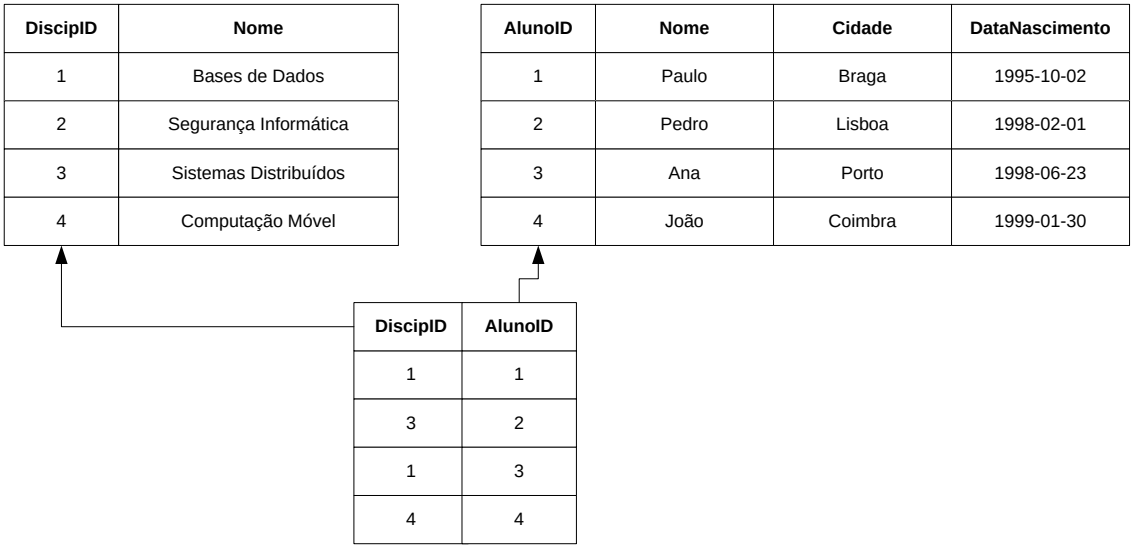


Figura 2-2 Exemplo de tabelas do modelo relacional.

A simplicidade do modelo relacional, e o seu fundamento teórico, contribuíram para a sua rápida adoção em todos os setores de atividade, sendo os SGBD atualmente praticamente ubíquos, e gerindo desde sistemas pequenos até sistemas com terabytes de informação.

2.2 Fatores de desempenho

Os SGBD foram desenvolvidos para gerir com eficiência as cargas de trabalho chamadas transacionais, em que muitas transações, de curta duração, acedem em modo de leitura e escrita a um conjunto significativo de dados. Estes SGBD processam cargas geralmente designadas por *On-Line Transaction Processing* (OLTP).

Os sistemas OLTP assumem um conjunto de compromissos que resultam das opções tomadas para desenhar a arquitetura dos primeiros SGBD:

- Como os dados são normalmente acedidos e modificados registo a registo, os dados devem ser armazenados em linha e cada registo deve ser atualizado por uma única operação de escrita. Além disso, os dados devem ser armazenados em páginas no disco que otimizam a quantidade de dados transferidos entre a memória e disco, e minimizam o número de acessos ao disco. Existe um compromisso entre o tamanho das páginas e a quantidade de dados que pode ficar bloqueada em acesso exclusivo durante uma transação.

- Os índices devem ser restritos aos atributos estritamente necessários para processar as consultas eficientemente, e para evitar o bloqueio excessivo do índice, assim, negar o acesso a conjuntos inteiros de linhas, que poderiam vir a ser necessárias para outras transações.
- A compressão de dados normalmente não é eficiente porque em cada linha há muitas vezes uma mistura de diferentes tipos de dados e valores que não estão relacionados. O tempo de processamento necessário para a compressão e descompressão não irá, portanto, ser recuperado por redução do volume de transferência de dados.
- Adicionar ou excluir atributos de índices é provável que seja demorado, uma vez que todas, ou uma grande parte, das páginas do índice podem ser afetadas, principalmente em caso de reorganização do índice, tornando-as inacessíveis a outras transações.
- As atualizações de um atributo, mesmo se for um predicado simples são suscetíveis de ser caras, porque toda a linha deve ser lida e escrita mesmo quando um único atributo for atualizado.

Adicionalmente, os sistemas OLTP têm um conjunto de requisitos que podem contribuir para penalizar o seu desempenho:

- A necessidade de manter um registo das modificações efetuadas para ser usado em caso de recuperação implica que as escritas estão associadas a mais escritas do *log* de recuperação. Embora as escritas para o log sejam otimizadas e efetuadas para discos dedicados e rápidos, o tempo de latência resultante da aplicação da regra WAL (*Write Ahead Logging*) é penalizador.

- A estratégia de gestão da memória-tampão pode não ser adequada a todos os tipos de transações, e pode efetuar substituições de páginas desadequadas ao conjunto de transações em curso. Por outro lado, esta estratégia não pode utilizar algoritmos muito complexos, e consequentemente demorados, para não ultrapassar o custo das leituras no disco.

- A execução de consultas é feita na memória dos clientes, implicando por vezes grandes volumes de tráfego entre o disco e a memória.

Para a manipulação analítica de dados, ou processamento analítico de dados (OLAP, *On-Line Analytical Processing*) verificou-se que essas regras não poderiam ser aplicadas sem afetar grandemente o desempenho. Neste contexto, houve a necessidade de mudar este conjunto de regras para satisfazer as necessidades de bases de dados de carácter analítico:

- As consultas tendem a utilizar apenas uma pequena parcela das colunas de uma relação, pelo que é vantajoso armazenar as relações por coluna. As colunas frequentemente lidas tenderão a ficar na memória-tampão e nas caches do processador, e a eficiência da cache será reforçada.

- Os dados são lidos mais vezes do que ser escritos ou atualizados, fazendo com que o tempo de CPU utilizado na criação de estruturas de armazenamento mais eficientes seja mais rentável. Como há poucas escritas, as páginas podem ser superiores ao tamanho de página que permitiria transferências atómicas entre o disco e a memória. Mesmo que uma leitura implique várias leituras de páginas físicas, não há risco de inconsistências devido a operações interrompidas a meio.

- Ao nível do controlo de concorrência, é possível dispensar as técnicas de bloqueio de páginas com fechos, uma vez que as escritas são muito poucas. É possível utilizar técnicas multiversão, em que cada consulta vê o estado de base de dados num ponto consistente no tempo, e em que não há bloqueios de páginas, ou muito poucos bloqueios.

- É possível indexar todos os atributos de uma relação, gerando assim índices chamados de cobertura, que evitam a leitura das linhas no disco, operando apenas nos valores encontrados nos índices.

- Ao ser armazenada na mesma página apenas uma coluna de cada linha, a compressão de dados é provável que apresente melhores resultados porque os dados são geralmente homogéneos e consequentemente os algoritmos de compressão dão bons resultados.

É no entanto possível melhorar o desempenho de sistemas OLTP relacionais tradicionais se alguns fatores forem alterados (Harizopoulos, Abadi *et al*, 2009):

- Nalgumas situações pode ser evitado o custo de escrever os *logs*, principalmente se a aplicação tolerar algum nível de inconsistência, ou se a replicação puder ser feita a partir de réplicas da base de dados.

- Nalgumas situações podem ser evitados mecanismos de controlo de concorrência pessimistas, permitindo um aumento de desempenho ao evitar a gestão de fechos e a contenção no acesso aos fechos.

Os mesmos autores chegam à conclusão, nas experiências que fizeram com um sistema tradicional, que apenas 6,8% do tempo total de processamento é realmente útil, sendo o resto gasto em *logs*, gestão de trincos, gestão de fechos, e gestão de memória tampão.

Vamos detalhar em particular um dos fatores de desempenho mais importantes, o tipo de armazenamento das relações, que tem consequências diretas na maior parcela dos custos de uma consulta, que é o acesso ao disco (Baykan, 2005). O desempenho está também diretamente relacionado com a capacidade de multiprocessamento das máquinas (Hardavellas, Pandis, Johnson, *et al*., 2006), mas neste trabalho não consideramos esse fator. A figura seguinte mostra os níveis de armazenamento num SGBD.

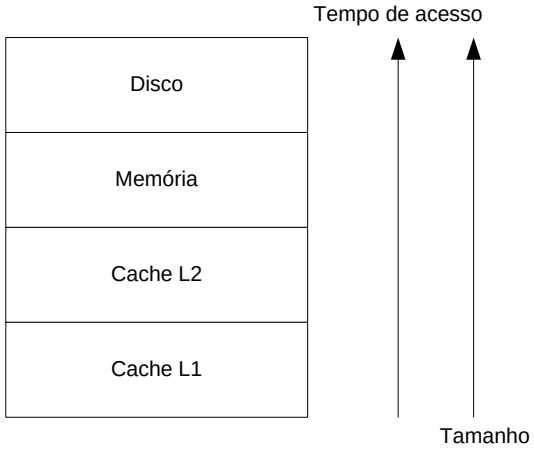


Figura 2-3. Níveis de armazenamento num SGBD.

Embora a transferência entre o disco e a memória constitua a maior parte do custo de execução de consultas (Deng, 2011), com o aumento da rapidez dos discos e do tamanho da memória esse fator passou para as caches L2 e L1, como demonstrado por Ailamaki, Dewitt, *et al* (1999); a taxa de sucesso na cache L2 foi apontado como o fator

mais importante na perda de desempenho de um SGBD. O tamanho da cache L1 é da ordem das dezenas de Kbytes, e da cache L2 pode ser um par de Mbytes. O tamanho da memória pode atingir dezenas de Gbyte. O comprimento de cada linha da cache é pequeno, da ordem dos 64 ou 128K. É importante que as linhas lidas da base de dados tenham um tamanho que lhes permita a utilização plena da cache (Ailamaki, DeWitt *et al*. 2001).

Vamos detalhar a seguir os modelos de armazenamento mais comuns, referindo quando útil o impacto que podem ter na gestão das caches L2 e L1.

2.3 Modelos de armazenamento

Um SGBD permite a distinção entre as propriedades físicas e conceptuais das tabelas da base de dados. Ao nível físico, as tabelas de base de dados precisam ser mapeadas numa estrutura de registos físicos, de comprimento fixo ou variável, que podem ser lidos facilmente conhecendo o deslocamento de cada um. Cada registo corresponde a informação de um tuplo da base de dados. Em disco, a informação é armazenada em páginas (também chamadas blocos) de tamanho fixo, geralmente múltiplo da unidade mínima de transferência entre o disco e a memória. São comuns tamanhos de página de 4k, 8k, 16k e 32k.

2.3.1 Modelo NSM

O formato mais comum de armazenamento é o formato N-ário (NSM, *N-ary Storage Model*), utilizado desde o protótipo **System R** (Astrahan, Blasgen, *et al* 1976). Nesta abordagem de armazenamento registo a registo mantém-se toda a informação sobre uma entidade junta, geralmente em blocos de armazenamento sequenciais no disco. Quando se lê um bloco de memória, lêem-se todos os registos armazenados nesse bloco. Esta abordagem pode ser ineficiente se os registos forem muito longos (consequentemente há menos registos por bloco), e se a consulta em questão só precisar de algumas colunas de cada registo (lê-se informação desnecessária que vai ocupar espaço de memória).

No modelo de armazenamento NSM cada bloco contém uma zona de apontadores para os registos localizados na página, o que facilita o seu endereçamento (como mostrado na figura seguinte). O endereço de um registo é constituído pelo endereço da página, e pela identificação da caixa que contém o apontador interno para o registo.

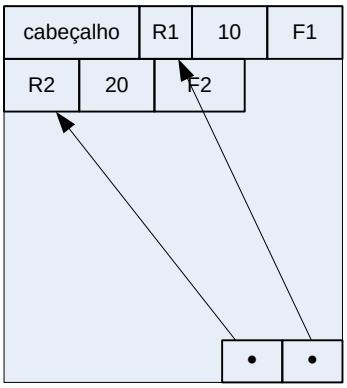


Figura 2-4. Bloco no modelo NSM[1]

Desta forma, os registos podem ser reorganizados dentro do bloco, por exemplo nos casos de inserção, remoção ou atualização, sem necessidade de alterar os endereços dos registos. Basta alterar o deslocamento dentro de cada apontador no fim do bloco. Se um registo é removido, coloca-se um marcador especial na casa do seu deslocamento, que pode assim ser reutilizado quando um novo registo de tamanho adequado for inserido na página. Não há necessidade nesse caso de reorganizar os apontadores. Na sua utilização mais comum, todos os registos numa mesma página NSM pertencem à mesma relação, embora haja sistemas, como Oracle, que permitam agrupar tuplos de relações diferentes numa mesma página quando se sabe que essas relações vão estar frequentemente envolvidas em junções.

2.3.2 Modelo DSM

Um modelo alternativo armazena os registos por coluna, mantendo no mesmo bloco apenas uma coluna de cada registo. Este modelo, conhecido por *Decomposition Storage Model* (DSM), foi proposto por Copeland e Khoshafian (1985). O modelo DSM transforma uma relação com N colunas em N relações binárias, e armazena apenas uma coluna por página, como mostrado na figura seguinte. As relações binárias contêm um identificador, que permite reconstruir os tuplos e se repete em todas as relações binárias, e uma coluna da relação original.

Os SGBD que permitem este tipo de armazenamento são geralmente conhecidos por SGBD verticais, ou SGBD em colunas.

[1] Figura inspirada de Gouveia (2014).

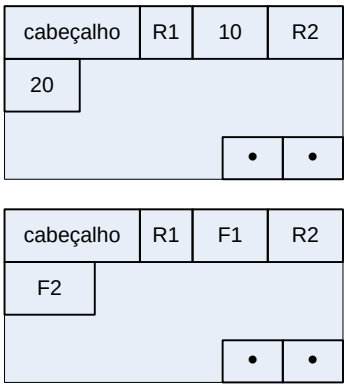


Figura 2-5. Bloco no modelo DSM.

Internamente uma página DSM pode estar organizada como NSM, com uma zona de apontadores para os registos. Numa dada página só existe uma coluna, o que permite um número de registos por página claramente superior ao modelo NSM. Assim, uma operação de leitura traz mais registos para memória, e potencialmente o número total de leituras numa dada consulta é reduzido.

O modelo DSM é atrativo em cenários onde as consultas acedem a apenas algumas colunas das relações; neste caso, cada bloco lido contém um número maior de relações binárias, diminuindo assim o número de acessos ao disco. Após os blocos serem lidos em memória, são efetuadas junções para reconstruir a relação a partir das relações binárias correspondentes que foram lidas. O modelo DSM otimiza o espaço memória ao evitar ler colunas que nunca serão usadas, ao contrário do NSM, que lê sempre todas as colunas de cada linha. Se no entanto uma consulta ler muitas colunas, o modelo DSM pode ter um desempenho inferior ao NSM, devido ao custo de leituras de mais páginas, e de reconstrução dos registos em memória. Khoshafian, Copeland *et al.* (1987) propuseram um processamento de consultas para DSM, chamado pivot, que podia tirar partido de processamento paralelo.

Estudos recentes, referidos acima, demonstraram que a utilização das caches do processador influenciam grandemente o tempo de execução de consultas. Neste caso, ao ler linhas mais pequenas (uma vez que só contêm uma coluna) é possível manter mais linhas em cache, e otimizar assim os acessos. Como as caches são pequenas e de comprimento fixo, também pequeno, se as linhas a guardar forem pequenas consegue-se otimizar a sua utilização.

Se considerarmos as operações de escrita, o modelo NSM acede e modifica o bloco onde se encontra o registo a modificar, criar ou remover. Estas operações são eficientes, podendo implicar reorganização do espaço, mas não implica modificar o endereço dos registos. O modelo DSM tem que aceder geralmente aos N blocos que contêm o registo a modificar ou remover; no caso de inserção, tem que fazer N escritas, uma para cada relação binária resultante da decomposição da relação inicial, e atualizar os índices para cada uma das entradas modificadas. No caso de relações com muitas colunas, estas operações podem penalizar o desempenho da base de dados.

Zukowski, Nes, e Boncz (2008) demonstraram que o tamanho dos tuplos influencia o desempenho de NSM e de DSM, ao permitir ou não o aproveitamento das caches L2 e L1. Em geral, NSM é superior a DSM quando se efetuam acessos aleatórios, por exemplo em junções e agregações. Os autores propõem também que a transformação entre NSM e DSM, sendo relativamente fácil e rápida, possa ser uma decisão do otimizador de consultas, optando por um modelo ou outro dependendo do tipo de consulta a processar.

2.3.3 Outros modelos

Outros modelos de armazenamento que foram sendo propostos incluem PAX (Ailamaki, DeWitt *et al*, 2002), Clotho (Shao, Schindler, *et al*, 2004), *Data Morphing* (Hankins, Patel, 2003) e *Fractured Mirrors* (Ramamurthy, Dewitt, Su, 2002). No entanto apenas o modelo tradicional NSM e mais tarde o DSM foram adotados por sistemas em produção. PAX era destinado especificamente a otimizar a utilização da cache, o que é cada vez mais importante nos processadores recentes. PAX utiliza mini-páginas DSM dentro de páginas NSM, aproveitando o melhor de cada modelo.

3 Modelos alternativos

3.1 Introdução

O modelo relacional, embora seja utilizado com sucesso na grande maioria das aplicações que necessitam de bases de dados, pode demonstrar-se inadequado em algumas aplicações, pelas seguintes razões:

- Tendo sido desenvolvido para uma arquitetura de computadores com mais de 40 anos, pode não tirar proveito dos recentes avanços da tecnologia.
- As aplicações diversificaram-se, e muitos tipos de aplicações diferem do cenário tradicional OLTP. OLAP, grafos, redes sociais, leilões, sensores, fluxo de dados em contínuo, são aplicações com requisitos que não são satisfatoriamente tratados no SGBD tradicional.

Uma abordagem consistiu em otimizar o modelo relacional, principalmente as possibilidades de processamento em *clusters* de partição de bases de dados, também conhecida por *sharding*, que permite escalar para dezenas, centenas ou milhares de bases de dados. As partições de bases de dados (chamadas *shards*) armazenam dados desnormalizados, de forma a evitar acessos entre partições. Quase todas as versões atuais dos SGBD mais usados permitem de alguma forma *sharding* automático, facilitando o escalamento de bases de dados conforme a carga de utilização aumenta.

Outra abordagem consiste em utilizar modelos diferentes do relacional, com arquiteturas e linguagens de manipulação e consulta diferentes. São esses modelos que vamos ver a seguir, começando pelos modelos NoSQL.

3.2 Modelos NoSQL

Entre os modelos alternativos ao relacional, os modelos ditos NoSQL têm tido muitos desenvolvimentos nos últimos anos. NoSQL é na realidade um termo genérico, utilizado de forma por vezes pouco precisa, que define uma classe de modelos de bases de dados, tendo apenas em comum o facto de não utilizarem SQL nem serem relacionais. A sua

arquitetura, modelo de dados, e linguagens de acesso e manipulação de dados podem ser as mais diversas. Com o aparecimento da computação em núvem, apareceram novos desafios e aplicações para estes modelos (Grolinger, Higashino *et al*., 2013). As características mais comuns dos modelos que se caracterizam como NoSQL são (Stonebraker, 2010):

- Esquemas simples e flexíveis, não-relacionais. Por vezes nem existe esquema, podendo ser possível guardar dados em qualquer estrutura.

- Possibilidade de escalar horizontalmente para qualquer número de servidores, quer em leitura, quer em escrita.

- Fornecer alta disponibilidade, garantindo as características AP do Teorema CAP (ver definição mais à frente). Isto significa que estes produtos sacrificam a consistência em favor da disponibilidade e da utilização de partições. Nos sistemas relacionais a ênfase é dada na consistência e na disponibilidade (CA).

- Ter características BASE (ver a seguir), em vez das características ACID dos sistemas relacionais clássicos.

Os modelos dos esquemas NoSQL mais comuns são (Cattel, 2010)(Oracle, 2012):

- Orientado a chaves: a estrutura desse modelo é como uma tabela *Hash*, ou seja, há diversas chaves na tabela, cada qual referencia um tipo de dados.

- Orientado a colunas: também chamado de modelo em colunas, os dados são armazenados em disco e agrupados em blocos por colunas. Na prática consiste em converter relações com N colunas em N relações binárias, tal como discutido no capítulo anterior para o modelo DSM.

- Orientado a documentos: similar ao modelo orientado a chaves, cada registo consiste num conjunto de pares chave-valor, e respetivos dados associados.

- Baseados em grafos: os dados são armazenados nos nós de um grafo cujas arestas representam o tipo de associação entre esses nós.

Alguns sistemas, como o Megastore do Google (Baker, Bond, Corbett, *et al*., 2011), oferecem simultaneamente características NoSQL e relacionais. Ao contrário dos

SGBD relacionais, que respeitam as características ACID, os modelos NoSQL respeitam as características conhecidas por BASE (Oracle, 2012):

- *Basically Available*: o sistema está sempre disponível, mesmo que temporariamente algumas partes não o estejam. Podem ser utilizadas partições, e replicação para garantir que o acesso aos dados é sempre possível.

- *Soft state*: os dados podem estar inconsistentes, e compete aos programadores de aplicações lidar com essas inconsistências.

- *Eventually consistent*: ao contrário dos SGBD relacionais, que verificam a consistência quando as transações confirmam, neste caso a única garantia é que num futuro próximo os dados estarão consistentes. Não há consistência instantânea.

As características BASE são necessárias quando as taxas de criação de novos dados e de processamento são muito elevadas, quando comparadas com as taxas de um sistema OLTP clássico, como um banco. Organizações como Facebook, Amazon, Google, LinkedIn, ebay, ou Twitter precisaram de soluções novas para lidar com um volume de tráfego sem precedentes. A maioria deu origem às soluções NoSQL mais usadas atualmente. Por exemplo, não é muito importante que um comentário apareça em todos os utilizadores ao mesmo tempo, podendo ser tolerados atrasos sem que isso provoque consequências graves. Como estes sistemas são distribuídos por centenas de servidores, adotam geralmente o que é conhecido pelo Teorema CAP (Gilbert e Lynch, 2002):

> Um sistema distribuído não consegue garantir sempre consistência, disponibilidade, e tolerância nas partições. Em qualquer instante, apenas duas destas três características são respeitadas.

No nosso estudo não incluímos as questões de sistemas distribuídos, porque isso iria implicar outros parâmetros de avaliação e exigiria uma estrutura que não estava disponível.

3.3 Modelos em coluna

O modelo de base de dados em coluna mantém cada coluna de base de dados separadamente, guardando continuamente os valores dos atributos pertencente à mesma

coluna de forma densa e comprimida. Esta forma de armazenamento pode beneficiar a leitura dos dados, porém, compromete a escrita em disco (Harizopoulos, Abadi, e Boncz, 2009). Abadi e Madden (Abadi e Madden, 2008), têm demostrado que os sistemas de base de dados verticais oferecem vantagens substanciais de desempenho, em relação a bases de dados tradicionais.

Com esta arquitetura, o valor de cada coluna (atributo) é armazenado em sequência, aumentando a desempenho da leitura de uma única coluna. Base de dados carrega em memória apenas os valores das colunas que serão utilizados, evitando preencher a memória com dados que não serão lidos ou utilizados. Há duas formas que as bases de dados têm de utilizar CPU para diminuir a utilização do disco.

- Codificação de dados para uma forma compacta (transformar os dados em bytes, ao envés de armazena-los em uma forma nativa).

- Armazenamento por coluna é mais simples para comprimir N valores, cada um de um tamanho K bits, em um espaço N*K bits (Stonebraker, Abadi, *et al* 2005).

A diferença fundamental entre armazenamento em coluna em relação a linha é o tipo de armazenamento. Armazenamento em linha, os tuplos são armazenados em linha na tabela de armazenamento enquanto, o armazenamento em coluna os tuplos passam a ser armazenados em colunas separadas. Essa forma de armazenamento pode beneficiar a leitura dos dados, porém comprometendo a escrita em disco. Deste modo, o armazenamento em coluna apresenta algumas vantagens em relação ao armazenamento em linha.

Uma das vantagens recai na capacidade de compressão de dados. Comprimir dados armazenados em linha, implicará que na mesma linha se encontram diversos valores de diferentes domínios. Comprimir os dados armazenados por coluna, implica que cada coluna irá conter os mesmos tipos de dados, tornando mais eficiente a compressão devido à homogeneidade dos dados, em relação a armazenamento em linha (Graeffe e Shapiro, 1991).

Os dados são suscetíveis de ser lidos muito mais frequentemente do que ser rescritos ou atualizados, o que proporciona a criação de estruturas para rentabilizar a CPU. Além disso os dados devem ser armazenados em grandes páginas de modo que grandes

números de itens relevantes possam ser recuperados numa única operação de leitura, resultando numa taxa de sucesso superior em geral.

O armazenamento por linha por outro lado, desfavorece o armazenamento em páginas grandes, já que cada operação de leitura de memória também lê os atributos que não são relevantes para a consulta em questão, desperdiçando largura de banda e espaço em memória, resultando assim numa baixa taxa de sucesso em geral.

O armazenamento por coluna também apresenta algumas desvantagens. Essas desvantagens recaem sobre a operação de escrita e construção dos tuplos. As desvantagens das operações de escrita devem-se às necessidades de separação dos atributos e seus componentes que serão armazenados separadamente, como também, a dificuldade de movimentar dados compactados dentro de uma página.

Sobre a operação de construção de linhas, apesar das informações de uma entidade lógica serem armazenadas em locais separados do disco, a maioria das consultas acede a mais do que um atributo de uma entidade, o que implica que dados de várias colunas têm de ser combinados em linhas, reconstruindo assim parte da estrutura dos tuplos originais (Abadi, Boncz & Harizopoulos, 2009).

Para armazenar os tuplos por colunas separadas é necessário que haja um relacionamento entre colunas que identifique que a mesma pertente a um tuplo específico. O método adotado na identificação inclui a inserção de uma coluna ID virtual para o armazenamento.

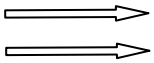
Otimizações importantes que melhoram o desempenho do SGBD orientado por colunas, recaem sobre as técnicas de materialização, indexação, compressão e diferentes estratégias para a execução de junções. Alguns autores propuseram técnicas para simular processamento por colunas em sistemas de armazenamento tradicionais, por linhas (El-Helw, Ross, Bhattacharjee, *et al.*, 2011). Holloway e DeWitt (2008) analisam os fatores de desempenho dos dois modelos e concluem que, sob determinadas condições de otimização, os modelos de armazenamento por linha podem oferecer desempenho comparável aos modelos por colunas. O número de predicados a testar e o número de colunas envolvidas numa consulta pode prejudicar em muito o desempenho dos modelos por colunas. Harizopoulos, Liang, Madden e Abadi (2006) testam o

desempenho de modelos em colunas e são mais otimistas quanto ao seu desempenho em condições desfavoráveis.

3.4 Otimização no armazenamento por coluna

Para armazenar as tabelas na base de dados, fisicamente as tabelas precisam de ser mapeadas numa estrutura de uma dimensão no espaço de memória do computador antes de serem armazenadas. O mapeamento das tabelas numa estrutura unidimensional pode ser feito por linhas ou por colunas.

No armazenamento em linha (do tipo NSM, como foi visto), os tuplos são armazenados em linha na tabela de armazenamento (como mostrado na figura seguinte):



Cod_Aluno	Nome	Idade	Cidade
1	Paulo	19	Braga
2	Pedro	20	Lisboa
3	Ana	30	Porto
4	João	23	Coimbra

Figura 3-1. Armazenamento por linha da tabela Aluno.

No armazenamento em coluna os tuplos passam a ser armazenados em colunas separadas (ver figura seguinte). Essa forma de armazenamento pode beneficiar a leitura dos dados, porém comprometendo a escrita em disco (Harizopoulos, Abadi, e Boncz, 2009) As operações de inserção e remoção de informação são típicas em sistemas de base de dados orientados por linhas. Porém, são mais difíceis de lidar, de forma eficiente, em sistemas de bases de dados orientados por colunas.

	Nome
1	Paulo
2	Pedro
3	Ana
4	João

	Idade
1	19
2	20
3	30
4	23

	Cidade
1	Braga
2	Lisboa
3	Porto
4	Coimbra

Figura 3-2. Armazenamento por coluna da tabela Aluno.

Além de cada linha ser dividida em diferentes colunas, antes de ser inserida, cada uma delas é armazenada separadamente e em diferentes locais do sistema de armazenamento. Porem, caso uma coluna seja comprimida e ordenada, a carga de trabalho numa operação de inserção torna-se bastante maior. Isto requer que cada linha seja dividida e

que sejam executadas mais operações de inserção que o habitual. Assim sendo, a abordagem comum é tentar atenuar o efeito causado pelas operações de inserção e, só depois, proceder à inserção em massa da informação.

Para melhorar a desempenho de sistema de base de dados orientadas por coluna, algumas técnicas são aplicadas, como materialização, indexação, compressão e técnicas de iteração em bloco. Estás técnicas serão descritas a seguir.

3.5 Compressão

Existem vários métodos de compressão, como LZ77 e LZ78 desenvolvidos por Lempel e Ziv (Matias, Labs, Hill, Rajpoot, 1998), algoritmos universais para compressão de dados. Estes algoritmos podem ser usados em base de dados, mas nem todos são aplicáveis em bases de dados verticais.

Intuitivamente os dados armazenados por coluna são mais compressíveis do que dados armazenados por linhas, devido essencialmente à homogeneidade dos dados. Comprimir utilizando algoritmos de compressão de dados específicos para bases de dados orientadas por colunas e operar com esses dados comprimidos melhora o desempenho das consultas até uma ordem de grandeza.

Os algoritmos de compressão têm melhor taxa de compressão com dados com baixa entropia de informação. Mesmo o espaço em disco ser barato, e cada vez mais barato, não impede que a utilização de compressão seja útil. A compressão melhora o desempenho (além de reduzir espaço em disco), pois se os dados são compactados e lidos a partir de disco para a memória (ou da memória para CPU) então menos tempo deverá ser gasto nas transferências entre o disco e a memória.

Na verdade, a compressão pode melhorar o desempenho de consulta além de simplesmente economizar em tempo de transferência de dados. Se o executor de uma consulta de uma base de dados orientada por coluna operar diretamente sobre dados compactados, a descompressão pode ser evitada e o desempenho pode ser melhorado. Além disso o método de armazenamento por coluna guarda os dados em coluna o que tornam mais aptas a compressão por informações serem armazenadas sequencialmente (Abadi e Madden, 2008).

A compressão de dados é um fator fundamental no desempenho dos SGBD, como também é um dos fatores na diferenciação na escolha de SGBD do modelo relacional ou modelo de armazenamento por coluna. Por exemplo, usando técnicas para comprimir cadeias de caracteres *Run-length Encoding* (RLE), onde existem sequências longas de caracteres repetidos, como AAAA, BBBBB é possível substituir sua representação de forma compacta por 4xA, 5xB. Desta forma operando sobre dados compactados o executor de consulta opera sobre múltiplos valores de uma coluna de uma só vez, e reduz os custos de CPU.

Quase todas as consultas podem operar diretamente sobre dados comprimidos, com a exceção de funções agregações (por exemplo não se podem somar os dados comprimidos), e os predicados que não sejam a igualdade. Nestes casos, os dados devem ser descomprimidos antes de serem usados, o que pode penalizar a utilização de técnicas de compressão.

3.6 Materialização

Ao armazenar dados por linha as informações sobre uma entidade lógica são armazenadas em vários locais no disco, o que não acontece com o armazenamento por coluna em que as informações sobre uma entidade lógica geralmente são colocadas em uma única linha da tabela. No entanto a maioria das consultas retorna mais do de um atributo de uma entidade, e consequentemente os dados de várias colunas contendo informações sobre a entidade devem ser combinados novamente em linhas de informações sobre essa entidade. Consequentemente, esta materialização de linhas (também chamada de "construção de linhas") é uma operação que penaliza o desempenho nos sistemas com armazenamento por coluna.

No armazenamento por coluna somente as colunas relevantes para uma dada consulta são lidas, são construídas linhas com esses atributos e são executadas operação de processamento de dados normais. Na realidade, num SGBD tradicional apenas o motor de armazenamento teria de ser mudado, um pouco à semelhança de MySQL que permite mudar o sistema de armazenamento (no caso mais comum permite optar entre MyISAM e InnoDB).

Abadi e Madden (Abadi e Madden, 2008) apresentam duas formas de materialização:

- Materialização precoce (*Early Materialization*), que consiste em adicionar uma coluna a uma linha intermediária de saída, caso a coluna lida seja requerida posteriormente por algum operador ou esteja incluída no resultado da consulta. Para uma dada consulta este método gera uma linha para cada linha da tabela em causa que poderá não aparecer no estado final.

- Materialização tardia (*Late Materialization*), que consiste em não adicionar a coluna no mesmo instante que é requerida. Assim, permitindo que o executor de consultas use operadores de alto desempenho para dados orientado a colunas de forma compacta, possivelmente permitindo-lhe construir menos tuplos.

A materialização precoce, sendo um método de construção de linhas no início de um plano de consulta deixa grande parte do potencial desempenho de bases de dados orientada a coluna não realizados ao partir muito rapidamente para um processamento de linhas (Abadi e Madden, 2008). O processo de execução da consulta é mostrado na figura seguinte:

<table>
<tr><td colspan="4">Armazenamento em linha

Tab. Funcionário-a</td><td colspan="3">Select nome, ano From Aluno Where Dia = 6 And Ano =2008;</td></tr>
<tr><td>Nome</td><td>Ano</td><td>Cidade</td><td>Dia</td><td rowspan="5">Construtor de tuplas =></td><td rowspan="5">Resposta=></td><td rowspan="5">Suzana</td></tr>
<tr><td>Ana</td><td>2005</td><td>Lisboa</td><td>9</td></tr>
<tr><td>Suzana</td><td>2008</td><td>Braga</td><td>6</td></tr>
<tr><td>Paulo</td><td>2005</td><td>Coimbra</td><td>6</td></tr>
<tr><td>Tiago</td><td>2010</td><td>Porto</td><td>4</td></tr>
</table>

Figura 3-3. Materialização precoce.

Na materialização tardia não são criados tuplo desnecessários devido à etapa de análise que permite selecionar apenas dados referentes às condições presentes na consulta.

Armazenamento em linha Tab. Aluno-a					Select nome, ano From Aluno Where Dia = 6 And Ano =2008;		
Nome	Ano	Cidade	Dia	Análise Prévia =>	Construtor de tuplas =>	Resposta=>	Suzana
Ana	2005	Lisboa	9				
Suzana	2008	Braga	6				
Paulo	2005	Coimbra	6				
Tiago	2010	Porto	4				

Figura 3-4. Materialização tardia.

Abadi, Myers, Dewitt e Madden (2007) realçam que, mesmo trabalhando com dados não comprimidos, a materialização tardia apresenta melhor resultados como mostra a figura seguinte[2].

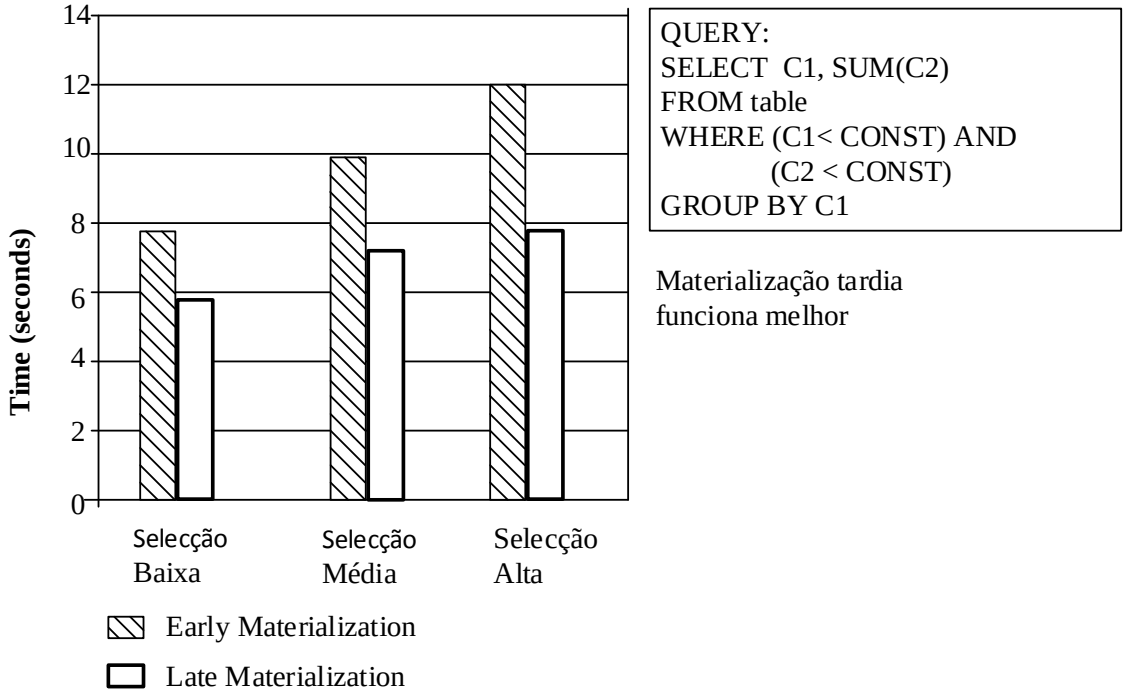


Figura 3-5. Comparação de tipos de materialização. Sem junções.

Esta avaliação foi feita sem a inclusão da junção, caso a junção fosse incluída o resultado seria o mostrado na figura seguinte[3]:

[2] Figura retirada de Abadi, Myers, Dewitt, & Madden (2007).
[3] Figura retirada de Abadi, Myers, Dewitt, & Madden (2007).

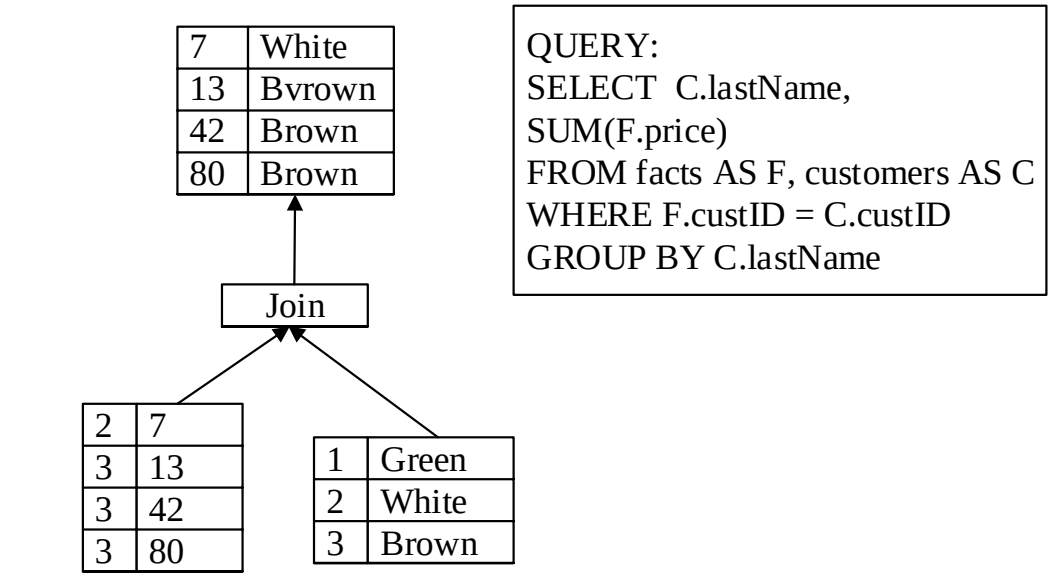


Figura 3-6. Materialização precoce. Com junção.

A figura seguinte detalha as etapas da materialização precoce no caso em que se usa uma junção.

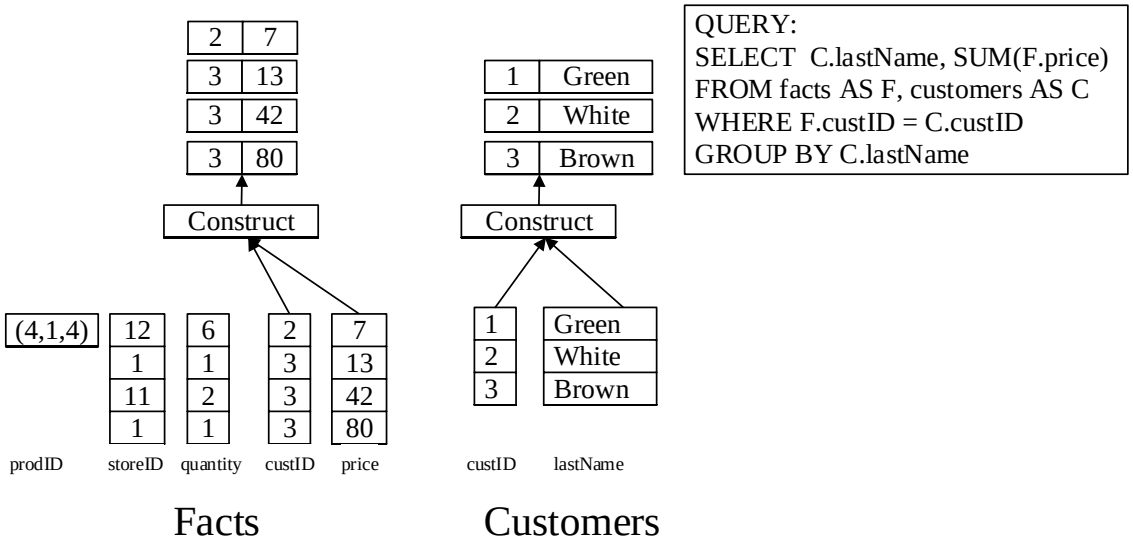


Figura 3-7. Materialização precoce. Com junção.

A materialização precoce apresenta os seguintes inconvenientes:

- Construção desnecessária de tuplos.
- Compressão precoce.
- Uso ineficiente da cache.

A materialização tardia visa melhorar essas fraquezas (Abadi, Myers, Dewitt e Madden, 2007). A figura seguinte mostra um exemplo de materialização tardia[4]:

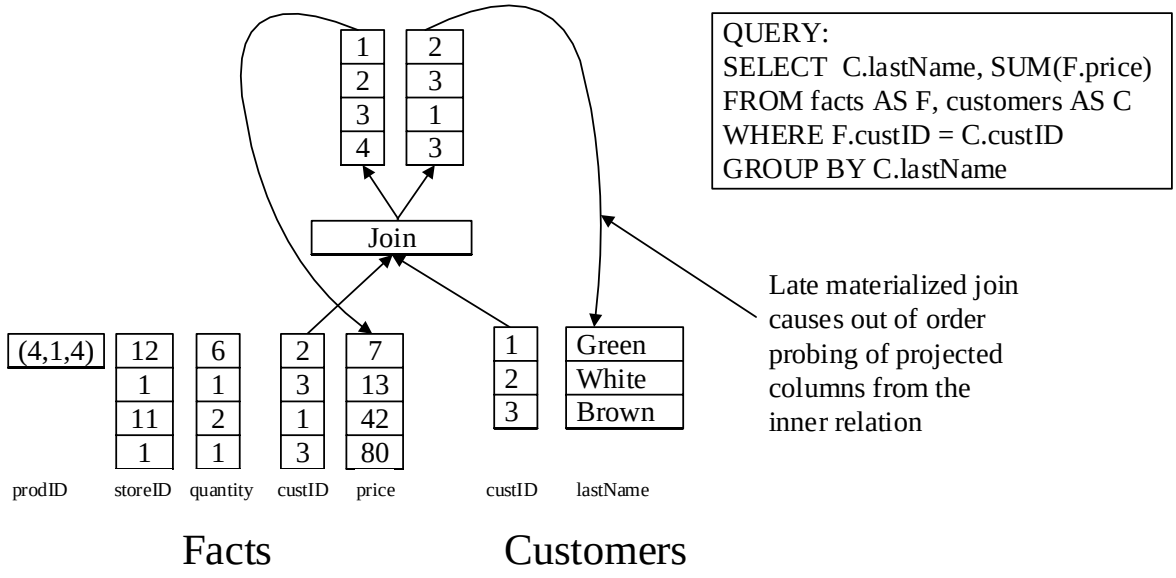


Figura 3-8. Materialização tardia. Com junção.

O desempenho de materialização tardia com junções é cerca de 2 vezes mais lento que a materialização precoce em relação ao desempenho das consultas (devido essencialmente à forma aleatória como são feitas as transferências de dados). Este resultado depende da quantidade de memória disponível, do número de atributos projetados e da cardinalidade de junção, mas esse resultado pode ser diferente caso seja usada a técnica de junções invisíveis (*Invisible Joins*) (Harizopoulos, Abadi, e Boncz, 2009).

Os mais recentes produtos com armazenamento por coluna como a versão X100 de MonetDB (Boncz, Zukowski e Nes, 2005), C-Store (Stonebraker, Abadi, Batkin, 2005) e a sua versão comercial Vertica, e Sybase IQ, optaram por manter os dados em colunas até muito mais tarde para o plano de consulta, que opera diretamente nestas colunas. Ao fazê-lo, as listas intermediárias de "posições" muitas vezes precisam de ser construídas a fim de se igualarem as operações que foram realizadas em diferentes listas de coluna, evitando que as junções mais tarde respeitem a mesma ordem dos elementos. Esta lista é enviada para a terceira coluna, para extrair os valores na posição desejada.

[4] Figura retirada de Abadi, Myers, Dewitt, & Madden (2007).

São cinco as vantagens de materialização tardia (Abadi et al., 2007):

1. Os operadores de seleção e de agregação tornam a construção de alguns tuplos desnecessária; se o executor esperar o tempo suficiente antes de construir um tuplo, pode ser capaz de evitar construí-lo completamente.

2. Se os dados são comprimidos através de um método de compressão orientado a coluna, deve ser descomprimido antes da combinação dos valores com os valores de outras colunas.

3. Os valores podem ser armazenados em memória de forma contígua em estruturas de dados orientadas por coluna, o que resulta em duas vantagens no desempenho: 1) a mesma técnica usada para comprimir dados armazenados por coluna em disco pode ser usada para comprimir os mesmos dados em memória, e 2) os ciclos de junção através de valores de estrutura de dados orientado por coluna tendem a ser mais rápidos do que os ciclos através de valores usando uma interface de iteração de tuplos.

4. O desempenho da cache é melhorado quando opera diretamente com os dados da coluna, uma vez que uma determinada linha de cache não desperdiça espaço com atributos irrelevantes para uma dada operação.

5. A otimização dos blocos tem um impacto sobre o desempenho para atributos de tamanho fixo. No armazenamento por linha se qualquer atributo de um tuplo for de tamanho variável, então todo o tuplo será de tamanho variável. Com a materialização tardia no armazenamento por coluna, as colunas de tamanho fixo podem ser operadas separadamente.

A materialização tardia pode sofrer várias otimizações, mas há situações onde pode ser mais penalizadora que a materialização precoce. Nos casos em que se usam predicados pouco restritivos em muitas colunas (por exemplo, “idade > 18 and sexo=’m’ numa base de dados de uma universidade, onde a grande maioria dos alunos terá mais de 18 anos), o esforço de intersectar muitas linhas de muitas colunas não compensa.

Abadi, Boncz *et al*. (2012) detalham o projeto e a implementação de bases de dados por colunas.

3.6.1 Iteração de Bloco

A fim de processar uma série de iterações de tuplos, o armazenamento em linha extrai os atributos necessários em cada tuplo através de uma interface de representação da mesma. Este processo necessita de tempo, percorrendo e interpretando cada tuplo e consumindo instruções da CPU para interpretar, antes de fornecer o resultado de uma consulta. Mesmo que o custo de leitura das páginas seja pequeno, o custo de CPU deve ser considerado no custo total da consulta, e é por vezes uma fração significativa deste.

Em contraste com a implementação de armazenamento em linha, os blocos de valores de armazenamento por coluna são enviados para um operador em uma única chamada de função. Além disso, não há necessidade de extração de atributos, e se a coluna é de largura fixa, esses valores podem ser repetidos diretamente como uma matriz. Operando em dados como uma matriz não só minimiza o tempo por tuplo, mas também explora o potencial de paralelismo em processadores modernos, como técnicas de *loop-pipelining* (Abadi e Madden, 2008).

3.7 Junção Invisível

Além das técnicas já apresentadas, como a compressão, a iteração de blocos e a materialização tardia, que muitas vezes são consideradas como técnicas básicas de otimização de bases de dados orientadas a coluna, há também técnicas de otimização avançadas como as junções invisíveis. Esta técnica surgiu para melhorar o desempenho das consultas na fase de agregação, ou seja o ponto fraco da materialização tardia em que a execução das consultas segue a ordem apresentada pelos eventos e aplica-se particularmente sobre o esquema estrela do modelo de armazém de dados; neste caso, restringe-se o conjunto de linhas da tabela de factos usando uma, ou mais, seleção do predicado na tabela dimensão.

A descrição desta técnica pode ser vista através do exemplo apresentado por Abadi (Abadi e Madden, 2008) . Essa execução entre a tabela facto e tabela dimensão, muitas vezes agrupada por atributos da tabela dimensão realiza-se para cada predicado selecionado.

No exemplo abaixo, pretende-se com a consulta encontrar a receita total de clientes que vivem na Ásia e que compraram produtos fornecido por um fornecedor Asiático entre

os anos de 1992 e 1997, agrupadas por combinação única da nação do cliente, da nação do fornecedor e do ano da transação.

```
SELECT c.nation, s.nation, d.year,
sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
supplier AS s, date AS d
WHERE lo.custkey = c.custkey
AND lo.suppkey = s.suppkey
AND lo.orderdate = d.datekey
AND c.region = 'ASIA'
AND s.region = 'ASIA'
AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

Na execução desta consulta cada predicado pode ser executado em paralelo e aplicar operações de bitmap e, alternativamente o resultado do predicado de uma aplicação pode ser direcionado a outra aplicação do predicado para reduzir o número de vezes que o segundo predicado deve ser aplicado. Após a aplicação de todos os predicados, são extraídas as linhas que contêm as informações relevantes para o predicado. Este processo de obtenção de informações dos predicados, após a execução de todos os predicados e que pode ser feito em paralelo, pode minimizar o número de extrações fora da ordem.

O operador de junção invisível é executado em três fases, descritas a seguir.

Na primeira fase cada predicado é aplicado à tabela dimensão correspondente e extraí do resultado a chave que satisfaz o predicado. A chave extraída é usada para construir uma tabela de *hash*, que pode ser utilizada para testar se um determinado valor da chave satisfaz o predicado como exemplifica a figura seguinte[5].

[5] Figura retirada de Abadi e Madden (2008).

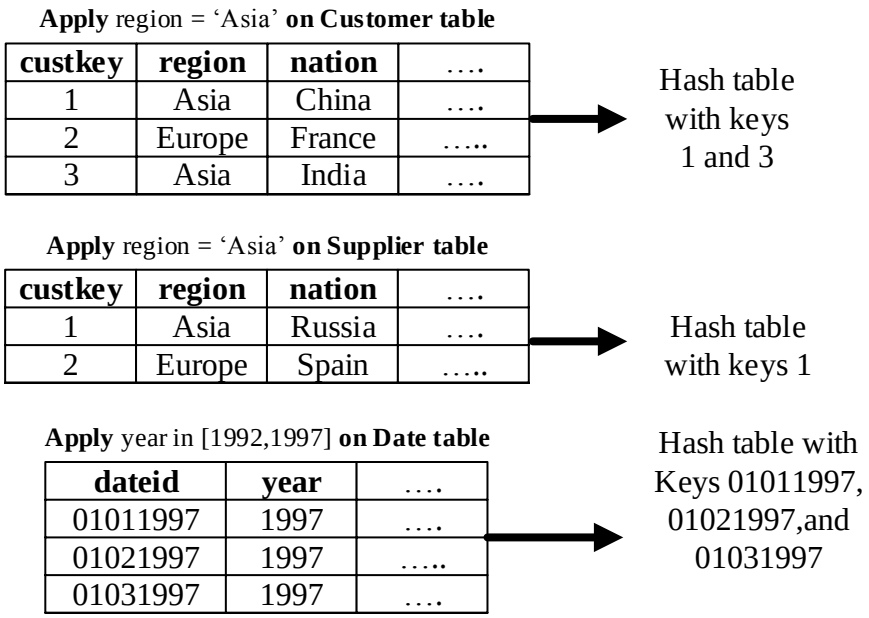


Figura 3-9. Primeira fase de junções para executar a consulta – filtros.

Na segunda fase, a tabela de hash criada é usada para extrair a posição do registo que corresponde ao predicado na tabela de factos. Este processo é feito através da interligação da tabela de hash com cada valor da chave estrangeira que satisfaze o predicado. Uma lista P é gerada (lista de posição de todos os predicados) que satisfaz a tabela de verdade como mostrado na figura seguinte[6].

[6] Figura retirada de Abadi e Madden (2008).

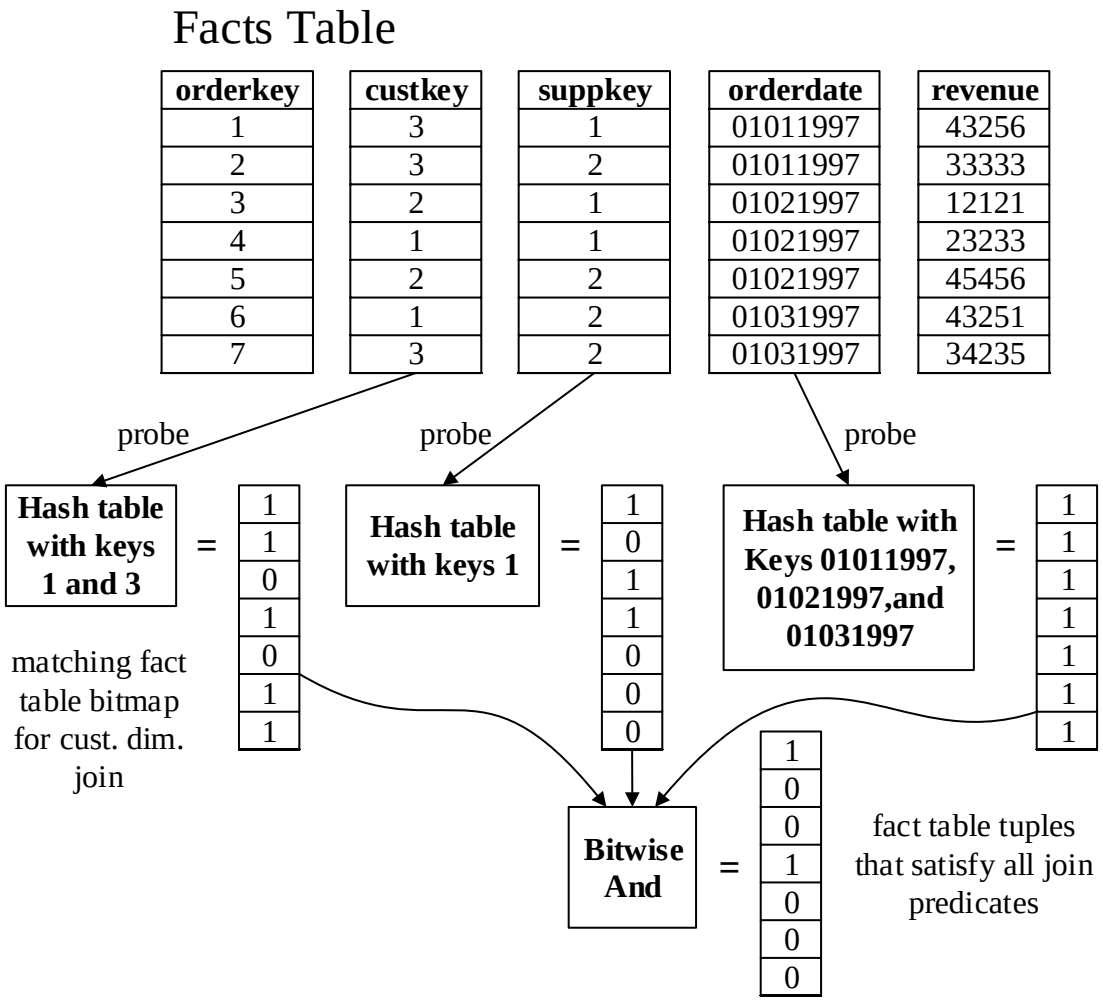


Figura 3-10. Segunda fase de junções – gera o resultado global do vetor P.

A terceira fase usa a lista que satisfaz a posição P criada na segunda fase. Para cada tabela dimensão existe uma coluna C de correspondência (uma chave estrangeira) na tabela facto para responder a consulta (por exemplo, clausula de referencia *group by*, ou função de agregação) e os valores da chave estrangeira de C são extraídos usando P. Esses valores apresentam a posição do valor desejado na coluna dimensão tornando a pesquisa muito mais rápida.

No exemplo apresentado a tabela 'orderdate' as chaves da coluna não são uma lista de identificadores de 0 e 1, portanto, deve-se realizar a junção em vez de apenas extrair a posição, como mostrado na figura seguinte[7].

[7] Figura retirada de Abadi e Madden (2008).

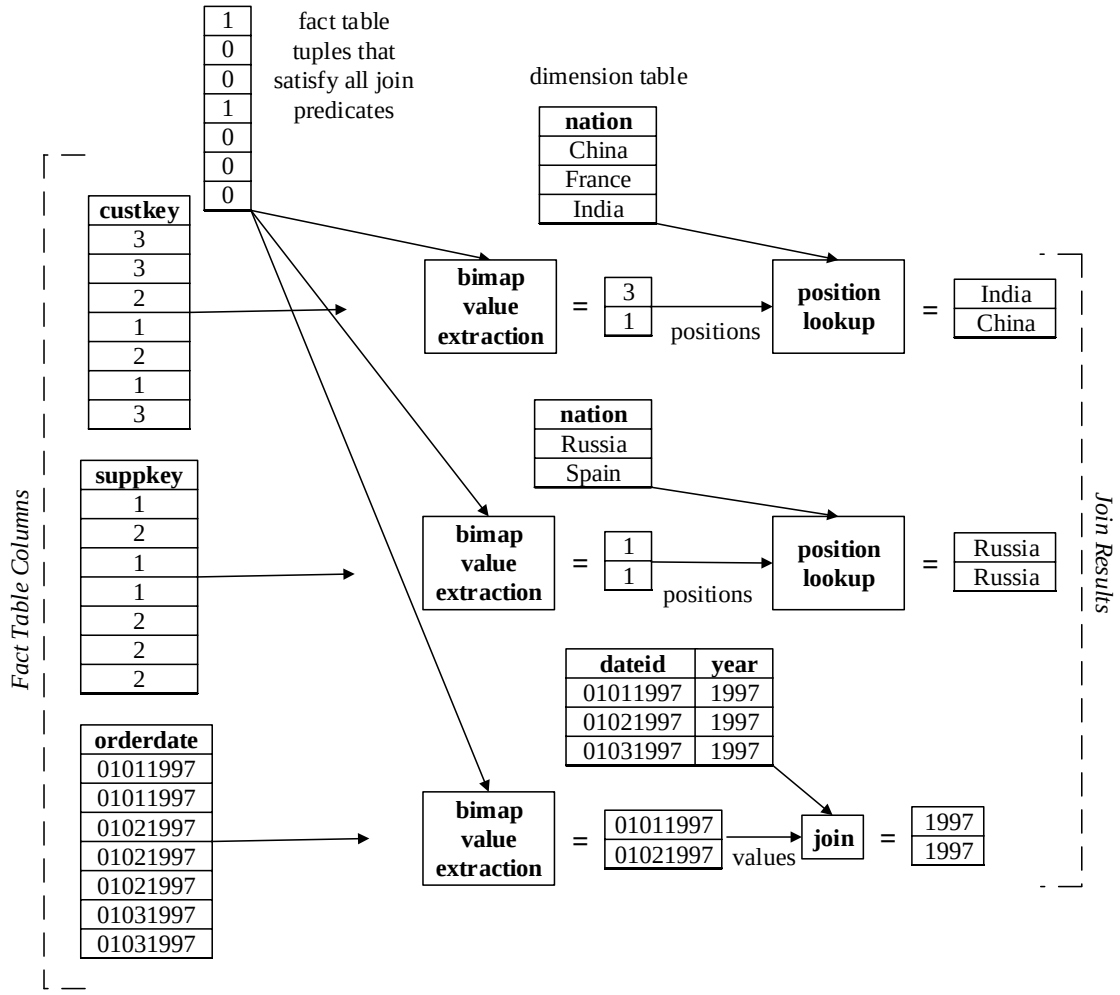


Figura 3-11. Terceira fase de junções – resultado de junções.

Nesta técnica de junção invisível o número de valores extraídos da tabela de factos é mínimo comparado com a materialização tardia porque a seleção é feita em toda a consulta e não apenas numa dimensão de cada vez (Abadi, Myers, Dewitt e Madden, 2007).

3.8 Aplicação de armazenamento por colunas

Com a evolução na arquitetura de computadores, e o aumento da capacidade do armazenamento não-volátil, tornou-se viável o uso do armazenamento nativo por coluna. Os sistemas mais indicados para o armazenamento por colunas são aqueles que precisam de ler e de realizar apenas operações de consulta em grandes volumes de dados, não sendo o mais indicados para o uso de sistemas que requerem operações de escrita. O aumento da capacidade de armazenamento (quer interno quer externo) foi muito superior ao aumento das velocidades de transferência dos discos, pelo que este continua a ser um fator importante na otimização.

Nos últimos anos houve um aumento das aplicações direcionadas para armazenamento em coluna, tais como (Harizopoulos, Abadi, e Boncz, 2009):

- Grandes armazéns de dados.
- Analises Personalizadas.
- Sistemas de exploração de dados (*data mining*).
- Aplicações de web semântica, com destaque RDF.
- Recuperação de informação em grande escala.
- Dados resultantes de grandes experiências científicas.

Contudo, a conclusão que podemos tirar das considerações mencionadas das análises feitas entre os dois modelos de armazenamento, é que o armazenamento em coluna se diferencia do modelo de armazenamento em linha por ser mais eficiente em determinadas situações. Em bases de dados com tabelas com várias centenas de colunas, em que apenas algumas são necessárias em cada consulta é importante minimizar o número de dados que se transferem e que não são relevantes para a consulta. Regras geral pode-se dizer que as aplicações analíticas têm a característica de aceder a poucas colunas de quase todos os tuplos, enquanto as aplicações transacionais têm a característica de aceder a muitas colunas de um número reduzido de tuplos.

O armazenamento por coluna possui vantagens ao utilizar compressão por ter a capacidade de organizar informações semelhantes de forma contígua; na materialização, por não precisar de processar tuplo desnecessários; na iteração em bloco pela capacidade de analisar todos os valores de uma coluna com um número menor de instruções de CPU e por ter a capacidade de organizar informações semelhantes de forma contígua. Já a nível de recuperação de dados, quando temos uma consulta que necessita de muitas colunas do mesmo tuplo ou quando temos tuplos pequenos que permitem que um grande número possa ser recuperado numa única instrução de acesso ao disco, o modelo de armazenamento por linha é superior ao modelo de armazenamento por coluna.

A partir dessas considerações, o modelo em colunas difere do modelo em linha por não seguir uma padronização base, torna-se interessante avaliar em que devemos optar por bases de dados em coluna em relação a bases de dados por linhas, tradicionais. Diversos SGBD com armazenamento em coluna possuem suporte para SQL e geram informações

de forma muito semelhante a outros SGBD com armazenamento orientado a linhas, podendo assim, haver uma comparação adequada entre os dois modelos.

3.9 SGBD por coluna

Existem vários SGBD com armazenamento nativo por colunas, entre os quais:

- MonetDB (deu origem à versão comercial Vectorwise).
- C-Store (deu origem à versão comercial Vertica).
- Sybase IQ.
- InfiniDB, compatível com MySQL.
- Infobright, compatível com MySQL.
- Greenplum, parcialmente baseado em PostgreSQL.
- CitusDB, integra com PostgreSQL.
- ParAccel.

Existem outros SGBD que, não sendo nativos por coluna, oferecem opções de armazenamento por coluna. Assim, SQL Server fornece, a partir da versão 2012, um índice por coluna que permite tirar partido de operações agrupando grandes quantidades de dados (Larson, Clinciu, Manson, *et al*., 2010). Citus Data oferece uma interface, cstore_fdw, que permite utilizar armazenamento por coluna em PostgreSQL. InfiniDB e Infobright oferecem motores de armazenamento vertical para MySQL. Oracle, a partir da versão 12c, oferece uma opção de armazenamento por coluna, paralelo ao sistema de armazenamento tradicional.

4 Armazéns de dados e modelos verticais

4.1 Introdução

Os armazéns de dados são cada vez mais comuns nas organizações, e o seu tamanho está na ordem dos terabytes, existindo organizações com peta bytes de informação. À medida que os históricos são guardados, e que cada vez há mais informação gerada, os acessos não transacionais, analíticos, são cada vez mais frequentes, e objeto de otimização.

Uma pequena comparação relacionada com as consultas das bases de dados transacionais e dos armazéns de dados, podemos dizer que no mundo transacional os objetivos do uso das bases de dados são em geral para automatizar as tarefas de negócios. Esses tipos de consultas tendem a ser consultas simples e pequenas (por exemplo, adicionar, remover, ou atualizar dados de um cliente, ou mesmo movimentação da conta bancária), e são consultas que não resultam de junções significativas dos dados. É comum estas consultas envolverem duas, três ou quatro tabelas.

A natureza das consultas nos armazéns de dados tende a ser menos previsível, de natureza exploratória e de natureza iterativa. Este tipo de consultas é mais duradouro, ou seja por serem de natureza analítica, os resultados não recaem só em uma única entidade como no modelo relacional, mas resultam da agregação de vários registos de várias entidades. As consultas são mais orientados a leitura do que a escrita, uma vez que após a recolha de informações e sua escrita no armazém de dados, em geral as consultas são somente de leitura, não modificando o conteúdo do armazém de dados. Por exemplo, pode-se querer conhecer a distribuição de vendas num dado período de tempo, por loja e por tipo de produto.

Os armazéns de dados são bases de dados que integram informações de diversas fontes, histórica, não voláteis, utilizadas para o armazenamento de dados visando auxiliar na tomada de decisão estratégica e, pode ser implementado usando as modelos em estrela ou floco de neve por meio do modelo relacional (Ralph e Margy, 2002). Tais esquemas

são compostos pelas tabelas de factos e de dimensões. Uma tabela de factos armazena as medidas quantitativas do negócio, enquanto uma tabela de dimensão caracteriza o assunto, por exemplo dimensão temporal como meses ou semestres, podendo os seus atributos formar hierarquias. O foco do nosso trabalho está direcionado para o esquema em estrela. O esquema floco de neve difere do esquema em estrela por normalizar as hierarquias dos atributos envolvidos nas tabelas de dimensão.

4.2 Estruturas de indexação

A estrutura de indexação é um fator importante para melhorar o desempenho no processamento das consultas. A utilização de índices *bitmap* é um dos métodos disponíveis de indexação mais eficientes para acelerar consultas, por permitir indexar valores multidimensionais e ser destinado essencialmente a atributos de leitura (Stockinger e Wu, 2007).

O índice *bitmap* de junção (O'Neil e Graefe, 1995) é comumente utilizado para processar consultas em armazéns de dados, e tem como principal vantagem a eliminação de junções entre as tabelas de factos e de dimensão. O índice bitmap é criado sobre os atributos das tabelas de dimensão, construindo-se vetores de bits para cada valor distinto dos atributos. O i-*ésimo* bit do vetor de bits armazenará o valor 1 se o valor correspondente ocorre no i-*ésimo* tuplo da tabela de fatos, caso contrário, o bit será 0.

Pelo facto de construir um vetor de bits para cada valor distinto, a cardinalidade dos atributos é um fator que pode diminuir a eficiência deste índice. Para melhorar o processamento, técnicas como codificação e compressão são utilizadas para contornar esse problema (O'Neil e Graefe, 1995)(Stockinger e Wu, 2007).

A figura seguinte mostra um exemplo de um índice bitmap para o atributo “sexo”.

ID	Sexo
1	M
2	M
3	F
4	M
5	N
6	F
7	F
8	M

Bitp 0 -F	Bitp 0 -M
0	1
0	1
1	0
0	1
0	0
1	0
1	0
0	1

M-Masculino.
F- Feminino
N- Não atribuído

Figura 4-1. Exemplo de um índice bitmap.

Podemos verificar na figura anterior como os dados são avaliados numa consulta em relação ao índice *bitmap*. As consultas são avaliadas com operações bit a bit lógico, o que é suportado de base pela eletrónica do computador, e é por isso extremamente eficiente. Para um atributo com determinados valores distintos, o índice de base bitmap gera os mesmos bitmaps com N bits distintos, onde N é o número de linhas no conjunto de dados. Cada bit em um bitmap é definido como "1" se o atributo no tuplo é de um valor específico; caso contrário, o bit é definido como "0".

4.3 Star Schema Benchmark

Nesta dissertação usaremos o esquema do teste *Star Schema Benchmark* SSBM para compararmos o desempenho de base de dados em colunas e tradicionais. A figura seguinte mostra um esquema em estrela composto pela tabela de factos **Lineorder** e pelas tabelas de dimensão **Customer**, **Supplie**r, **Date**, e **Part** (O'Neil, O'Neil, Chen, 2009). A dimensão de cada tabela é dada para um fator de escala de 10.

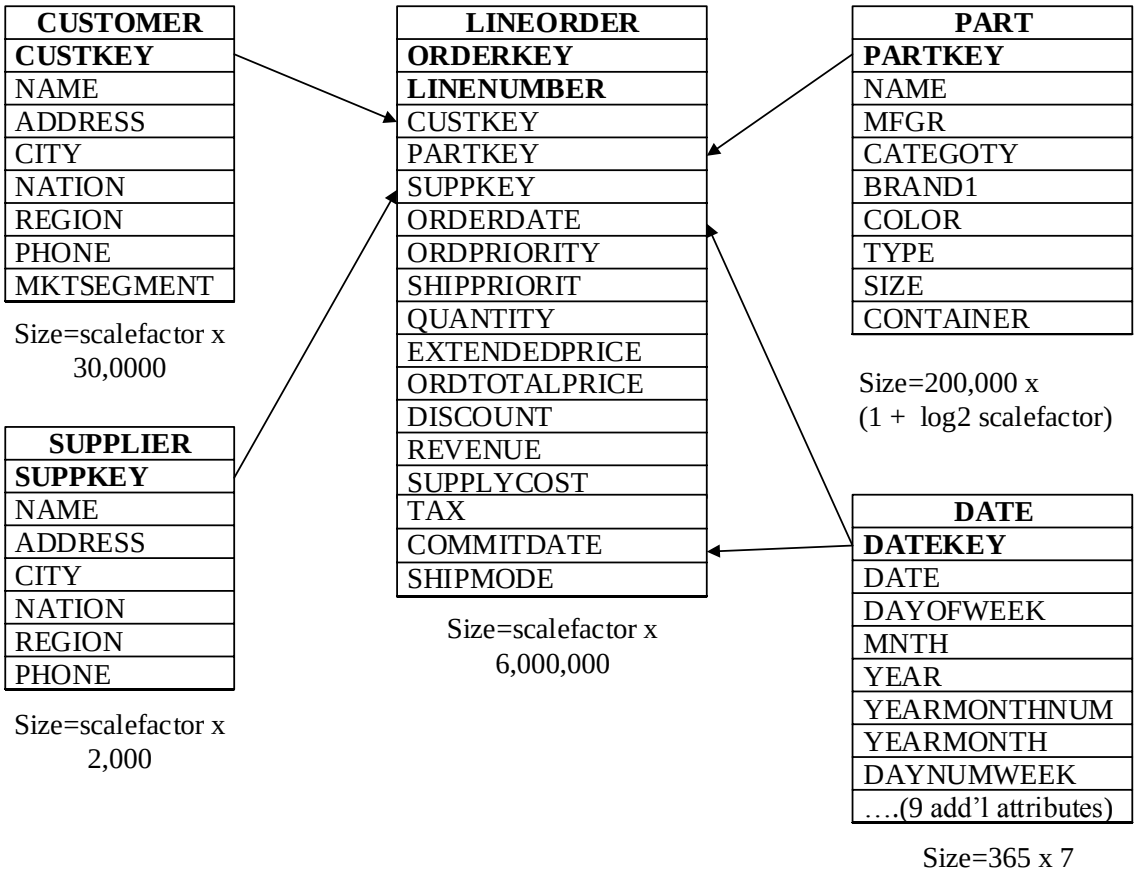


Figura 4-2. Esquema SSBM

O SSBM foi projetado para medir o desempenho dos SGBD com funcionalidades de armazém de dados, e é baseado no teste TPC-H (TPC, 2014). O teste TPC-H baseia-se num cenário de sistemas de apoio à decisão, tenta ser o mais realista possível, e consiste num conjunto de consultas arbitrárias que simulam o acesso a um armazém de dados de encomendas e vendas de produtos.

O esquema usado no teste é composto por uma tabela de factos, a tabela LINEORDER que combina com as tabelas de TPC-H, ou seja tabelas LINEITEM e ORDERS. Há 17 colunas na tabela LINEORDER que contêm informação individual dos pedidos. É definida uma chave primária composta, composto pelos atributos ORDERKEY e LINENUMBER. Os atributos chave estrangeira da tabela LINEORDER incluem referências para as tabelas CUSTOMER, PART, SUPPLIER, e DATE, bem como a ordem dos atributos, incluindo a sua prioridade quantidade, preço, desconto e outros atributos.

As tabelas de dimensão contêm informações sobre as respetivas entidades. Os atributos *region*, *nation*, *city* e *address* da tabela dimensão **Supplie**r, constituem uma hierarquia

geográfica permitindo a agregação de dados e o processamento de consultas *drill-down* (obter mais detalhe numa dada dimensão) e *roll-up* (agregar e subir na dimensão, reduzindo o seu número) geralmente usadas em aplicações OLAP.

Este esquema em estrela difere do esquema floco de neve. O esquema floco de neve normaliza as hierarquias envolvidas, aumentando consequentemente o número de junção envolvidas.

4.4 MonetDB

Nos últimos anos tem aumentado o interesse e as aplicações referentes a bases de dados em coluna. Esse aumento deve-se às possibilidades de desempenho superior demonstradas pelas bases de dados orientadas por coluna em relação a linha, com referência a determinadas cargas de trabalho. Embora os resultados apresentados por vários autores demostrem claramente a superioridade do armazenamento em coluna em relação ao tradicional, por linha, também é possível que uma modelação diferente nos modelos tradicionais resulte num desempenho semelhante ao desempenho dos modelos em coluna. Stonebraker *et al*. (2007) defendem que o SGBD tradicional tem quarenta anos e está baseado em considerações de arquitetura de computadores que já não existem, em necessidades de processamento que foram há muito ultrapassadas, e que por isso é necessário passar do SGBD "todo o serviço" para SGBD personalizados para as diferentes tarefas e desafios que surgem.

Para comparar o armazenamento por coluna, nativo, com os produtos tradicionais, escolhemos MonetDB, um dos primeiros produtos a fornecer armazenamento por coluna (Manegold, Kersten e Boncz, 2009).

4.4.1 Estrutura e Arquitetura básica

MonetDB é um SGBD de código aberto, e gratuito, para aplicações de alto desempenho em exploração de dados, OLAP, bases de dados científicos, consultas com XML, consultas em texto integral, e consultas multimédia, que está a ser desenvolvido desde 1993 pelo grupo de investigação de arquitetura de base de dados do CWI em Amsterdão (Groffen e Nes, 2012). O aparecimento de bases de dados científicas com quantidades enormes de dados (por exemplo no caso do genoma humano, ou da exploração do

espaço) coloca novos desafios aos sistemas de gestão de bases de dados (Svensson, Boncz, Ivanova, Kersten, Nes, 2009)(Ivanova, Nes, Gonçalves, Kersten, 2007).

MonetDB foi um dos pioneiros a fornecer bases de dados orientadas a coluna que usa compressão a fim de reduzir o tráfego do disco. Porém a principal motivação do uso deste modelo não se limitou a reduzir I/O do disco em pesquisas em bases de dados científicas ou em consulta de análise de informação de negócios.

A arquitetura de MonetDB foi baseada em outras considerações dadas no modelo original Decomposed Storage Model (DSM), ou seja, é focado de álgebra de consulta de armazenamento dos dados, com objetivo de alcançar maior eficiência da CPU. Além do eficiente padrão de acesso vertical, MonetDB emprega uma série de técnicas para proporcionar melhor desempenho para aplicações analíticas. Entre elas, estão a otimização de tempo de execução.

MonetDB está projetado para explorar a eficiência e a eficácia da memória durante o processamento de consultas em computadores com processadores recentes e o seu principal foco de consulta predomina a leitura e a atualização concentrando-se na carga de trabalho analítica e científica (Groffen e Nes, 2012). O projeto também abrange a flexibilidade em vários níveis pela utilização de módulos de extensão, implementados em C ou na linguagem MAL de MonetDB. É possível a criação de novos algoritmos, o que facilita o suporte a requisitos com carácter especial para além do SQL. Usa fragmentação vertical para armazenar as tabelas, e os dados são armazenados exclusivamente numa estrutura denominada *Binary Association Tables* (BAT) que será apresentada mais à frente.

A arquitetura de MonetDB dispõe-se em 3 camadas, a camada exterior, a camada interior e a camada do núcleo que têm os seus próprios meios de otimização e correlacionam-se entre si (Groffen e Nes, 2012).

A camada exterior é a camada superior. Esta camada de aplicação oferece interfaces de consulta para SQL, XQuery (linguagem para consulta de dados no XML), SciQL, e SPARQL. As consultas são analisadas em representações específicas de domínio, como álgebra relacional para SQL. A estratégia específica de otimização tem o principal objetivo de reduzir o tamanho de dados a ser processados. Os planos de execução lógica

gerados são então traduzidos em instruções de *Assembly MonetDB Language* (MAL), que são passados para a próxima camada.

A camada média ou interior consiste em estruturas de optimizadores MAL e compilador MAL de interface textual para o núcleo. Cada optimizador transforma um determinado programa MAL em um programa mais eficiente, eventualmente adicionando diretrizes de gestão de recursos. Esta tática de otimização é mais inspirada pela otimização de linguagens de programação que pela otimização clássica de consultas de bases de dados. Difere dos optimizadores baseados em custos, reconhecendo que nem todas as decisões podem ser traduzidas em uma única fórmula de custo. Operando com álgebra relacional binária, estes módulos do optimizador são partilhados por todos os modelos de dados e pelas linguagens de consulta.

A camada do núcleo ou inferior fornece o modelo da tabela binaria, onde todas as tabelas consistem exatamente de duas colunas. Essas tabelas são chamadas de *Binary Association Tables* (BAT). Essa camada permite que operadores da álgebra executem otimizações operacionais, ou seja, no período de execução escolher o algoritmo e a implementação a ser usado com base nas propriedades das entradas.

A arquitetura de MonetDB apresenta algumas variantes para casos específicos que requerem grandes processamentos de dados como em aplicações de exploração de dados, armazéns de dados, e processamento de XML por exemplo. Algumas dessas variantes, como MonetDB/X100, são recomendadas para aplicações OLAP. MonetDB Servidor é recomendado para soluções de servidor de bases de dados MonetDB, MonetDB/XQuery para soluções de Bases de dados XML, MonetDB /GIS para gráficos, e MonetDB/SQL para base de dados relacionais.

Este modelo de dados, projetado para trabalhar com arquitetura exterior/interior alcançou a máximo nível de desempenho, devido ao seu projeto suportar as várias extensões dos modelos lógicos (Boncz, 2002). MonetDB permite também organização automática das colunas que podem ser segmentadas de forma adaptativa, dependendo das características da consulta (Ivanova, Kersten e Nes, 2008).

A figura seguinte[8] representa a arquitetura exterior/interior de MonetDB onde a camada interior representa o núcleo central onde são instalados as bases de dados para as múltiplas interfaces. A camada exterior mapeia as consultas de bases de dados (por exemplo no acesso a bases de dados relacionais), e pode mapear aplicações muito específicas.

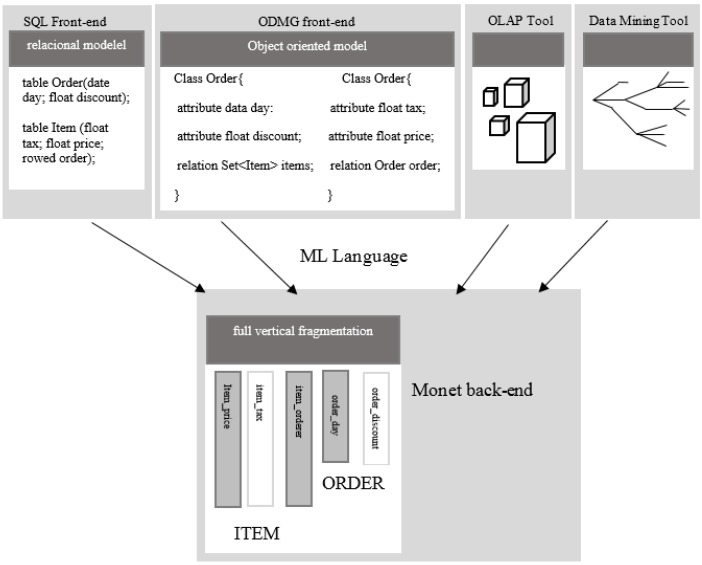


Figura 4-3. Arquitetura de MonetDB.

A extensibilidade e desempenho de MonetDB devem-se essencialmente às seguintes características (Boncz, 2002):

- Uso de tabelas binárias para armazenamento de dados, e sua forma de manipulação.
- Possibilidade de utilizar vários tipos de paralelismo.
- Disponibilização de uma linguagem de acesso procedimental e computacionalmente completa.

MonetDB tem outras características importantes, e uma grande extensibilidade para domínios específicos que lidam com grandes volumes de dados.

[8] Figura retirada de (Boncz, 2002).

4.4.2 Indexação

O método de indexação mais comum que é utilizado na maioria das bases de dados é a B+-Tree. O seu uso é de grande importância devido a sua eficácia no processamento das transações. Este método de indexação tem a mesma complexidade operacional para consultas e atualizações de índices. Desta forma para a maioria das aplicações de armazéns de dados e processamento de consultas analíticas (OLAP), as características de atualização de índices não se aplicam, pois as operações de consulta são mais frequentes do que as de atualização (Stockinger e Wu, 2007).

O índice bitmap básico usa cada valor distinto do atributo indexado como uma chave, e gera um bitmap que contém o maior número de bits como o número de tuplos no conjunto de dados para cada chave. Existem três estratégias básicas para reduzir o tamanho dos índices bitmaps (Stockinger e Wu, 2007):

- Codificação.
- Compressão.
- Agrupamento (*binning*).

Oracle usa a estrutura de índice BITMAP para otimização de consultas nas tabelas que possuem colunas com baixa cardinalidade, ou seja, colunas que possuem pouca variação de valores nas linhas de uma tabela. O índice só será utilizado pelo optimizador de Oracle quando se verifica que é mais rápido fazer um varrimento de índice (*Index Scan*) do que realizar um varrimento de tabela (*Full Table Scan*). O índice BITMAP é criado somente se a tabela sofre poucas variações, caso contrário pode implicar contenção no acesso a tabelas indexadas que sofrem atualizações frequentes.

Nas secções seguintes vamos apresentar sucintamente os outros dois produtos que foram selecionados e utilizados nas experiências. São produtos gratuitos, de código fonte aberto, e considerados como tendo bom desempenho e respeito pelas especificações de um SGBD relacional. É igualmente importante referir que não impõem restrições à divulgação de resultados de testes, ao contrário de outros SGBD que não autorizam a divulgação desses resultados.

4.5 PostgreSQL

O SGBD PostgreSQL é um sistema relacional conforme à norma ANSI SQL, e que oferece um conjunto completo de funcionalidades para gestão de bases de dados. Tem como características principais o facto de permitir extensões do tipo objeto-relacional, por exemplo autorizando herança entre relações, e um sistema de armazenamento multiversão.

PostgreSQL oferece algumas características consideradas avançadas, tais como tipos definidos pelo utilizador, indexação geométrica, índices parciais e com funções, e índices definidos pelo utilizador (GiST). Por exemplo, o índice geométrico R-Tree é na realidade um índice GiST definido para indexar dados bidimensionais.

PostgreSQL 9.4 foi o primeiro SGBD multiversão a respeitar o nível máximo de serialização previsto na norma ANSI SQL. O mecanismo de controlo de concorrência de PostgreSQL é o *Snapshot Isolation*, o que favorece as cargas de leitura, não utilizando fechos e permitindo leituras de versões anteriores se necessário. PostgreSQL utiliza fechos, de uma forma pessimista, para implementar determinados níveis de isolamento.

Como PostgreSQL pode ser otimizado para cargas de leitura, evitando fechos, foi considerado um bom candidato para participar nestas experiências, uma vez que tem um armazenamento NSM clássico e um sistema de controlo de concorrência que fornece um bom desempenho com cargas de leitura.

4.6 CitusDB

CitusDB é uma extensão do PostgreSQL 9.4 permitindo a utilização de partições e o aumento de bases de dados a serem utilizadas numa determinada instalação. O CitusDB *Community Edition* está disponível gratuitamente. CitusDB permite a utilização de uma base de dados vertical, em simultâneo com a de PostgreSQL, inspirada no formato *Optimized Row Columnar* (ORC), ele próprio uma extensão do formato RCFile desenvolvido pelo Facebook. As características principais do formato ORC são:

- Compressão, o que permite reduzir de um fator de 4 a 6 vezes o espaço ocupado em disco e na memória.

- Projeção de colunas, o que permite ler apenas as colunas que são relevantes para uma dada consulta.

- Índices de intervalos, o que permite guardar valores máximos e mínimos e assim evitar leituras de valores que caiam fora desses intervalos.

O armazenamento por colunas usa a interface de programação de PostgreSQL, o que permite utilizar sempre a versão mais recente, e permite também a criação de novos tipos de dados e a utilização de todos os tipos de dados permitidos por PostgreSQL.

Embora baseado em PostgreSQL, CitusDB foi considerado um bom candidato para participar nestas experiências, uma vez que fornece otimização para grandes quantidades de dados não disponíveis em PostgreSQL, e é anunciado como sendo adaptado a processamento analítico de grandes quantidades de dados. CitusDB pode ser usado para substituir o PostgreSQL sem fazer quaisquer alterações na camada de aplicação e permite manter uma alta disponibilidade devido ao seu sistema de recuperação de falhas.

A arquitetura de CitusDB esta dividida em dois nós, como representa a figura seguinte. O nó mestre e os nós trabalhadores. Os nós trabalhadores são adicionados no arquivo de filiação do no mestre ou principal. O utilizador interage com o nó principal através da interface padrão do PostgreSQL para consultas e carregar dados. O nó mestre armazena apenas os meta dados e os dados são distribuídos entre os nós trabalhadores.

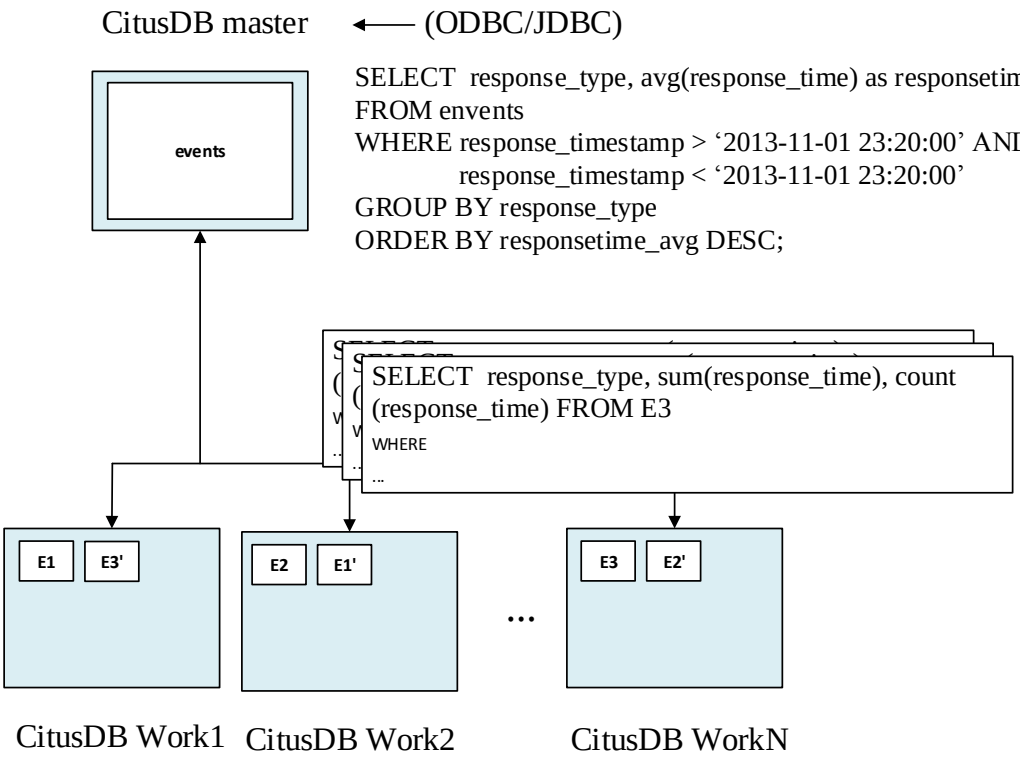


Figura 4-4. Arquitetura de CitusDB.

CitusDB utiliza uma arquitetura em bloco de forma modular semelhante ao Hadoop Distributed File System (HDFS), mas diferencia-se na utilização de tabelas de partição horizontal nos nós trabalhadores em vez de ficheiros.

Cada fragmento é replicado pelo menos em dois dos nós do *cluster*, com isso a perda de um único nó não terá impacto sobre a disponibilidade dos dados. A arquitetura lógica *sharding* do CitusDB permite também adicionar novos nós em qualquer momento para aumentar a capacidade e poder de processamento do *cluster*.

Cada tabela em CitusDB tem exatamente uma coluna que é escolhida como a coluna de distribuição, que informa a base de dados sobre a estatística de distribuição de cada fragmento. Assim o optimizador de consultas distribuídas do CitusDB aproveita essas colunas para determinar a melhor forma de execução da consulta. CitusDB apresenta dois padrões de consultas que variam de acordo com o tipo de consulta em uso: a distribuição por tempo e a distribuição por identificador. A distribuição por identificador pode ser benéfica quando a consulta envolve filtros e junção.

O CitusDB suporta equi-junções entre qualquer número de tabelas, independentemente do seu tamanho e método de distribuição. O gestor de consultas escolhe com base nas estatísticas recolhidas o melhor método de junção a ser utilizado.

Nestas experiencias não foram alteradas quaisquer configuração do CitusDB sendo usado a sua configuração padrão na comparação dos resultados.

5 Análise de desempenho

5.1 Introdução

Foi feita a análise experimental de desempenho nas plataformas dos sistemas MonetDB5, PostgreSQL 9.4 e CitusDB da CitusData, em que foi usada uma base de dados gerada por um fator de escala a partir do teste SSBM (TPC-H).

O desempenho dos sistemas de bases de dados reflete-se na capacidade de processamento de múltiplas consultas. O resultado do processamento sofre influência do tamanho da base de dados na qual as consultas são executadas, da velocidade de transferência do disco, e do número de acessos quando as consultas são submetidas por múltiplas transações concorrentes. Existem vários trabalhos que comparam, sob diferentes condições, o desempenho de modelos por colunas e tradicionais (Stonebraker, Bear, Çetintemel *et al*., 2007) (Shannon e Benninger, 2014).

A seguir descrevemos alguns dados específicos de SSBM para atingir o nosso objetivo, e não fazemos uma descrição detalhadas de TPC-H, que pode ser consultada em (TPC, 2014).

5.2 Base de dados utilizada

A base de dados foi gerada a partir do executável DBGEN. DBGEN é um programa de geração de dados para de base de dados experimentais, escrito em ANSI ‘C’ e utilizado em TPC-H. Para gerar os dados com o DBGEN, usa-se a linha de comando para gerar ficheiros de texto ASCII, que podem posteriormente ser importados por diferentes SGBD.

Cada ficheiro contém dados separados por delimitadores para as tabelas definidas na base de dados TPC-H. As tabelas contêm dados gerados por um fator de escala escolhido com o formato <table.tbl> (por exemplo, Customer.tbl). As tabelas SSBM criadas são mostradas a seguir:

```
create table CUSTOMER (       C_CUSTKEY int, C_NAME string,
```

```
C_ADDRESS string,
C_CITY string,
C_NATION string,
C_REGION string,
C_PHONE string,
C_MKTSEGMENT string,
CONSTRAINT pk_customer_custkey
 PRIMARY KEY (c_custkey));
```

```
P_BRAND1 string,
P_COLOR  string,
P_TYPE string,
P_SIZE int,
P_CONTAINER string,
CONSTRAINT pk_part_partkey
PRIMARY KEY (p_partkey));
```

```
create table SUPPLIER (
S_SUPPKEY int,
S_NAME string,
S_ADDRESS string,
S_CITY string,
S_NATION string,
S_REGION string,
S_PHONE string
CONSTRAINT pk_supplier_suppkey
PRIMARY KEY (s_suppkey));
```

```
create table PART (
P_PARTKEY int,
P_NAME string,
P_MFGR string,
P_CATEGORY string,
```

```
create table DWDATE (
D_DATEKEY int,
D_DATE string,
D_DAYOFWEEK string,
D_MONTH string,
D_YEAR int,
D_YEARMONTHNUM int,
D_YEARMONTH string,
D_DAYNUMINWEEK int,
D_DAYNUMINMONTH int,
D_DAYNUMINYEAR int,
D_MONTHNUMINYEAR int,
D_WEEKNUMINYEAR int,
D_SELLINGSEASON string,
D_LASTDAYINWEEKFL int,
```

```
D_LASTDAYINMONTHFL int,
D_HOLIDAYFL int,
D_WEEKDAYFL int
CONSTRAINT pk date datekey
PRIMARY KEY (d_datekey));

create table LINEORDER (
LO_ORDERKEY int,
LO_LINENUMBER int,
LO_CUSTKEY int,
LO_PARTKEY int,
LO_SUPPKEY int,
LO_ORDERDATE int,
LO_ORDERPRIORITY string,
LO_SHIPPRIORITY string,
LO_QUANTITY int,
LO_EXTENDEDPRICE int,
LO_ORDTOTALPRICE int,
LO_DISCOUNT int,
LO_REVENUE int,
LO_SUPPLYCOST int,
LO_TAX int,
LO_COMMITDATE int,
LO_SHIPMODE string;
CONSTRAINT
pk_lineorder_orderkey_linenumbe
r PRIMARY KEY (lo_orderkey,
lo_linenumber),
CONSTRAINT
fk_lineorder_commitdate_date_da
tekey FOREIGN KEY
(lo_commitdate) REFERENCES
dwdate (d_datekey),
CONSTRAINT
fk_lineorder_custkey_customer_c
ustkey FOREIGN KEY (lo_custkey)
REFERENCES customer
(c_custkey),
CONSTRAINT
fk_lineorder_orderdate_date_dat
ekey FOREIGN KEY (lo_orderdate)
REFERENCES dwdate (d_datekey),
CONSTRAINT
fk_lineorder_partkey_part_partk
ey
FOREIGN KEY (lo_partkey)
REFERENCES part (p_partkey),
CONSTRAINT
fk_lineorder_suppkey_supplier_s
uppkey FOREIGN KEY (lo_suppkey)
 REFERENCES supplier
(s_suppkey));
```

As tabelas de dimensão foram geradas de acordo com os dados mostrados a seguir para um fator de escala (SF) unitário.

Para gerar o SF = 1 (1GB), para popular a tabela, usamos: dbgen -vfF -s 1

Para gerar versões com um SF = 1 (1GB), usamos: dbgen -v -U 1-s 1

dbgen -s 1 -T c	part.tbl
dbgen -s 1 -T p	supplier.tbl
dbgen -s 1 -T s	date.tbl
dbgen -s 1 -T d	Tabela de factos lineorder.tbl
dbgen -s 1 -T l	Para todas as tabelas SSBM

As 13 consultas SSBM usadas para analisar o desempenho dos diferentes SGBD são listadas no Anexo A.

5.3 Condições experimentais

As experiências foram feitas num computador portátil com sistema operativo Ubuntu 15.0, 64 bit, Processador Intel Core i5 (CPU M430, 2.27GHz), e 4GB de memória RAM. O computador tinha várias outras aplicações e processos a correr em simultâneo, não estando dedicado em exclusivo para as experiências.

Os testes realizados tiveram um fator de escala de 1GB e 2GB, resultando em duas bases de dados. Os conjuntos de dados foram executados nos sistemas de bases de dados MonetDB5, CitusDB e PostgreSQL 9.4 nas mesmas condições. Cada base de dados foi executada seis vezes sobre as consultas acima mencionadas, calculando-se assim a média dos resultados. O resultado das experiências é descrito na secção seguinte.

5.4 Resultados experimentais

Para a primeira base de dados (1GB, denominada agora TPC-H1), os resultados, em segundos, foram os seguintes:

TPC-H1			
	PostgreSQL	CitusDB	MonetDB
Q1	2,614	2,221	1,100
Q2	2,427	2,093	0,254
Q3	2,326	2,045	0,222
Q4	3,437	3,234	0,424
Q5	2,183	2,066	0,308
Q6	1,732	1,468	0,268
Q7	3,376	2,905	0,481
Q8	1,882	1,757	0,314
Q9	2,006	1,946	0,250
Q10	2,079	1,783	0,242
Q11	4,064	3,462	1,000
Q12	3,275	3,169	0,551
Q13	2,829	1,922	0,359

Tabela 1: TCP-H1 tabela de valores experimentais

Pode verificar-se que a consulta 11 (Q11) é a que apresenta, dentro de cada SGBD, a maior variação, demorando aproximadamente o dobro do tempo de qualquer uma das outras consultas. A consulta 12 apresenta um padrão semelhante. Em MonetDB5, a consulta Q1 é a mais demorada, o que não acontece com os outros SGBD, onde demora aproximadamente o mesmo tempo que as outras consultas.

A figura seguinte mostra o gráfico dos valores obtidos na tabela anterior.

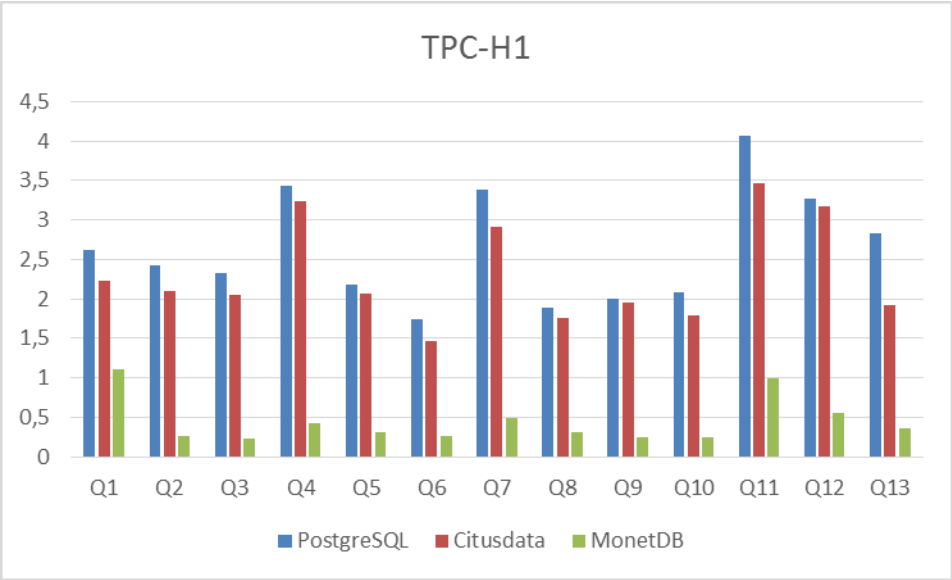


Figura 5-1. TCP-H1: gráfico dos valores experimentais.

Para a segunda base de dados (2GB, denominada agora TPC-H2), os resultados, em segundos, foram os seguintes:

TPC-H2			
	PostgreSQL	CitusDB	MonetDB
Q1	47,304	19,360	6,100
Q2	13,183	14,237	0,356
Q3	6,625	6,625	0,330
Q4	10,073	8,531	3,600
Q5	5,982	3,972	0,507
Q6	3,432	3,432	0,439
Q7	7,642	7,618	1,900
Q8	3,794	3,296	0,469
Q9	3,880	3,308	0,376
Q10	3,629	3,254	0,388
Q11	8,581	7,651	3,100
Q12	6,260	5,784	1,100
Q13	14,347	4,540	0,596

Tabela 2: TCP-H2 tabela de valores experimentais

Neste segundo conjunto de dados nota-se o aumento expressivo dos tempos de resposta de PostgreSQL e de CitusDB, enquanto a maior dos valores de MonetDB5 sofreu ligeiros acréscimos.

A figura seguinte mostra o gráfico dos valores obtidos na tabela anterior.

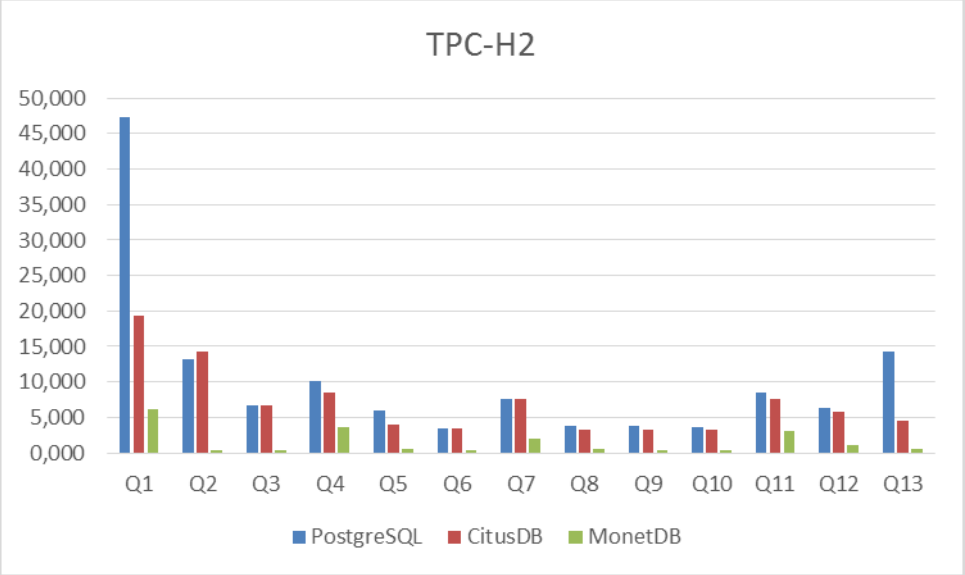


Figura 5-2. TCP-H2: gráfico dos valores experimentais.

Os tempos de resposta das consultas usadas na fase experimental variam de acordo com as complexidades de consultas. As comparações de bases de dados em relação ao desempenho estão relacionadas com os tempos gastos nas realizações de consultas, ou seja, tempo gasto de I/O juntamente com o tempo gasto nas execuções das consultas.

Há pouca variações de complexidades referente as consultas comuns o que reduz os cálculos de I/O e portanto o impacto referente a tempos a esses tipos de consultas são menores. As consultas que utilizam filtros, junções, ou outras funções complexas aumentam o tempo de execuções de consultas, o que afeta significativamente o tempo global de I/O.

Os resultados experimentais demostram que o uso de técnicas de junções e materialização melhoram as bases de dados em coluna em relação as base de dados relacionais.

Estes resultados demostram que as bases de dados em coluna superam as bases de dados relacionais, o que resulta na redução dos tempos de I/O para cargas de trabalhos de consultas analíticas.

6 Conclusões

Este trabalho focou-se no estudo comparativo do desempenho de bases de dados por coluna em relação a base de dados tradicionais, com armazenamento por linha.

A experiência foi feita com base de dados geradas sistematicamente a partir do teste TPC-H com um fator de escala de 1GB e um fator de escala de 2GB, usando os sistemas de gestão de base de dados PostgreSQL, CitusDB e MonetDB5. Todas as experiências foram executadas num computador portátil, não dedicado em exclusivo e com várias aplicações e processos a correrem em simultâneo.

Foram testadas as 13 consultas do teste SSBM (ver Anexo A) e os resultados dos tempos de execução das consultas foram recolhidos nas mesmas condições e características iguais para todos os sistemas de avaliação. Cada consulta foi executada seis vezes e retirou-se a média dos valores, devido à flutuação de I/O e à utilização da cache, o que poderia influenciar a avaliação do desempenho.

Os resultados experimentais demonstraram que, para cargas do mesmo tipo que as utilizadas no teste SSBM, essencialmente analíticas, o desempenho da base de dados em coluna MonetDB5 supera as bases de dados tradicionais, como PostgreSQL e CitusDB. As técnicas aplicadas de junção e materialização na consulta influenciaram os resultados, uma vez que a materialização filtra subconjuntos da tabela de factos relacionados com os predicados das consultas e, por vezes, evita o cálculo de junções com as tabelas dimensão. Em relação ao espaço de armazenamento ocupado pelas bases de dados experimentais verificou-se que devido ao método de compressão usado por MonetDB5, a base de dados ocupou um espaço menor que o PostgreSQL, uma vez que este último SGBD não usa qualquer tipo de compressão.

Podemos dizer que o CitusDB apresenta o melhor desempenho em relação a PostgreSQL e, que esse desempenho está relacionado com a forma como esses dados são armazenados. Desta forma CitusDB ganha no desempenho no carregamento dos dados no disco e perde em relação ao MonetDB5 que além de possuir o armazenamento por coluna, possui o processamento de consultas baseadas em coluna.

Não foi possível testar no âmbito experimental bases de dados maiores que 2GB, o que poderia dar mais informação sobre a variação do desempenho com o volume de dados em disco.

O objetivo principal desta dissertação propunha avaliar o desempenho de base de dados em relação ao tempo de execução. Contudo, apesar dos resultados obtidos nos diversos testes experimentais serem bastante interessantes e demostrarem o melhor desempenho de MonetDB, em relação as outras bases de dados em estudo, esse resultado é muito restrito dentro dos conjuntos das bases de dados. Seria interessante modificar os parâmetros originais de cada produto, por forma a sintonizar o melhor possível para cada carga de transações.

A realização de um estudo mais aprofundado poderia retirar outras informações sobre os sistemas avaliados, como por exemplo, um estudo mais alargado sobre a comparação dos dois modelos avaliados, meios experimentais mais adequados, e criação de índices de armazenamento por coluna nos sistemas de bases de dados tradicionais que o permitissem.

Por último, podemos referir que nem todos os SGBD permitem a publicitação de resultados de testes, o que torna difícil, e limita muito, a escolha dos produtos a comparar.

Bibliografia

Abadi, D. J., Boncz, P. A., Harizopoulos, S. (2009). Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2), p. 1664–1665.

Abadi, D., Boncz, P., A. Harizopoulos, S., Idreos, S., Madden, S. (2012). "The Design and Implementation of Modern Column-Oriented Database Systems", *Foundations and Trends® in Databases*, 5(3), p. 197–280.

Abadi, D. J., Madden, S. R. (2008). Column-Stores vs . Row-Stores : How Different Are They Really ? *Proceedings of SIGMOD 2008*, Vancouver, BC, Canada.

Abadi, D. J., Myers, D. S., Dewitt, D. J., Madden, S. R. (2007). Materialization Strategies in a Column-Oriented DBMS, *Proceedings of ICDE 2007*, Istambul, Turquia.

Ailamaki, A., Dewitt, D., *et al* (1999). *DBMSs on a modern processor: Where does time go?*, University of Wisconsin Computer Sciences Technical Report 1394.

Ailamaki, A., DeWitt, D., Hill, M., Skounakis, M. (2001). Weaving Relations for Cache Performance, *Proceedings of the 27th VLDB Conference*, Roma, Itália.

Ailamaki, A., D. J. DeWitt et al (2002). "Data page layouts for relational databases on deep memory hierarchies", The VLDB Journal 11.

Astrahan, M., Blasgen, M. *et al* (1976). "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems* 1(2).

Baker, J., Bond, C., Corbett, J. et al. (2011). Megastore: Providing Scalable, Highly Available Storage for Interactive Services, *Proceedings of CIDR 2011*, Asilomar, California.

Baykan, E. (2005). "Recent Research on Database System Performance", Technical Report, École Polytechnique Fédérale de Lausanne, Suiça.

Boncz, P.A. (2002). *Monet: A next-Generation DBMS Kernel For Query-Intensive Applications*, Tese de doutoramento, Universidade de Amsterdão, Amsterdão, Holanda.

Boncz, P.A., Zukowski, M., Nes, N. (2005) MonetDB/X100: Hyper-Pipelining Query Execution, *Proceedings of the* 2005 CIDR Conference, Asilomar, California.

Cattel, R., (2010). "Scalable SQL and NoSQL Data Stores", *ACM SIGMOD Record*, 39(4).

Codd, E. F. (1970). “A relational model of data for large shared data banks”, *Communications of the ACM*, 13(6), p. 377-387.

Copeland, G. P., S. N. Khoshafian (1985). “A Decomposition Storage Model”, *ACM SIGMOD Record* 14(4).

Deng, Y. (2011). "What is the Future of Disk Drives, Death or Rebirth?", *ACM Computing Surveys* 43(3).

El-Helw, A., Ross, K., Bhattacharjee, B., *et al*. (2011). Column-Oriented Query Processing for Row Stores, *Proceedings of DOLAP'11*, Glasgow, Escócia.

Gilbert S, Lynch N (2002). Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33(2).

Gouveia, Feliz (2014). *Fundamentos de Bases de Dados*, 1ª ed., FCA, Lisboa.

Graeffe, G., Shapiro, L. (1991). Data Compression and Database Performance, *Proceedings of ACM/IEEE-CS Symposium on Applied Computing*, Kansas City.

Groffen, F., Nes, N. (2012). "MonetDB : Two Decades of Research in Column-oriented Database Architectures", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, p. 1–6.

Grolinger, K., Higashino, W., Tiwari, A., Capretz, M. (2013). "Data management in cloud environments: NoSQL and NewSQL data stores", *Journal of Cloud Computing: Advances, Systems and Applications*.

Hankins, R., Patel, J. (2003). Data Morphing: An Adaptive, Cache-Conscious Storage, *Proceedings of the 29th VLDB Conference*, Berlin.

Harizopoulos, S., Liang, V., Madden, S., Abadi, D. (2006). Performance Tradeoffs in Read-Optimized Databases, *Proceedings of VLDB'06*, Seul, Coreia.

Harizopoulos, S., Abadi, D., Boncz, P. A. (2009). "Column-Oriented Database Systems", Tutorial VLDB 2009.

Hardavellas, N., Pandis, I., Johnson, R. *et al*. (2006). "An Analysis of Database System Performance on Chip Multiprocessors", Relatório Técnico CMU-CS-06-153, Carnegie Mellon University.

Hellerstein, J., Stonebraker, M. et al (2007). “Architecture of a Database System”, Foundations and Trends® in Databases, 1(2).

Holloway, A., DeWitt, D. (2008). Read-Optimized Databases, In Depth, *Proceedings of VLDB'08*, Auckland, Nova Zelândia.

Ivanova, M., Nes, N., Gonçalves, R., Kersten, M. (2007). MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database, *Proceedings of 19th*

International Conference on Scientific and Statistical Database Management (SSDBM 2007), Calgary, Canada.

Ivanova, M., Kersten, M., Nes, N. (2008). Self-organizing Strategies for a Column-store Data, *Proceedings of EDBT'08*, Nantes, França.

Khoshafian, S., Copeland, G., et al. (1987). A query processing strategy for the decomposed storage model, *Proceedings of the Third International Conference on Data Engineering*, Los Angeles, California.

Larson, P.-A., Clinciu, C., Manson, E., *et al*. (2010). SQL Server Column Store Indexes, *Proceedings of SIGMOD'10*, Atenas, Grécia.

Manegold, S., Kersten, M., Boncz, P. A. (2009). Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct, *Proceedings of VLDB'09*, Lyon, França.

Matias, Y., Labs, B., Hill, M., & Rajpoot, N. (1998). "Implementation and Experimental Evaluation of Flexible Parsing for Dynamic Dictionary Based Data Compression", *Proceedings of WAE98*, Saarbrucken, Alemanha, p. 1–3.

O'Neil, P., Graefe, G. (1995). "Multi-Table Joins Through Bitmapped Join Indices", *ACM SIGMOD Record*, 24(3), p. 8–11.

O'Neil, P., Neil, B. O., Chen, X. (2009). "Star Schema Benchmark, revision 3", disponível em http://www.cs.umb.edu/~poneil/StarSchemaB.pdf.

Oracle (2012). Oracle NoSQL Database, Oracle White Paper, Oracle Corporation, Redwood Shores, California.

Ralph, K., Margy, R. (2002). *The data warehouse toolkit: the complete guide to dimensional modeling*, Wiley.

Shao, M., Schindler, J. *et al* (2004). Clotho: Decoupling Memory Page Layout from Storage Organization, *Proceedings of the 30th VLDB Conference*, Toronto, Canada.

Ramamurthy, R., Dewitt, D., Su, Q. (2002). A Case for Fractured Mirrors, *Proceedings of VLDB*, Hong Kong, China.

Shannon, M., Benninger, C, (2014). "Telemetry Database Query Performance Review", SophosLabs, Network Security Group.

Stockinger, K., & Wu, K. (2007). Bitmap Indices for Data Warehouses, *Capítulo em* Wrembel R., Koncilia Ch., (2007) *Data Warehouses and OLAP: Concepts, Architectures and Solutions.* Idea Group, Inc.

Stonebraker, M. (2010). SQL Databases v. noSQL Databases, Blog@ACM, *Communications of the ACM* 53(4).

Stonebraker, M. Abadi, D., Batkin, A., *et al.* (2005). C-Store: A Column-oriented DBMS, *Proceedings of the 31st VLDB Conference*, Trondheim, Noruega.

Stonebraker, M., Bear, C., Çentimel, U., Cherniack, M. *et al*. (2007). One Size Fits All? – Part 2: Benchmarking Results, *Proceedings of 3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California.

Stonebraker, M., Maden, S., Abadi, D., Harizopoulos, S., *et al.* (2007). The End of an Architectural Era (It's Time for a Complete Rewrite), *Proceedings of VLDB'07*, Viena, Áustria.

Svensson, P., Boncz, P. A., Ivanova, M., Kersten, M., Nes, N. (2009). Emerging "vertical" database systems in support of scientific data, *Capítulo em* Scientific Data Management: Challenges, Technology, and Deployment, Shoshani, A., Rotem, D. eds, Chapman & All / CRC.

TPC (2014). TPC Benchmark^{TM} H Standard Specification Revision 2.17.1, Transaction Processing Performance Council (TPC), S. Francisco, California.

Zukowski, M., Nes, N., Boncz, P. A. (2008). DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing, *Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008)*, Vancouver, Canadá.

Anexo A Consultas SSBM

```
-- Consulta 1:
select sum(lo_extendedprice*lo_discount) as revenue
       from lineorder, dwdate
       where lo_orderdate = d_datekey
              and d_year = 1993
              and lo_discount between 1 and 3
              and lo_quantity < 25;
-- Consulta 2:
select sum(lo_extendedprice*lo_discount) as revenue
       from lineorder, dwdate
       where lo_orderdate = d_datekey
              and d_yearmonthnum = 199401
              and lo_discount between 4 and 6
              and lo_quantity between 26 and 35;
-- Consulta 3:
select sum(lo_extendedprice*lo_discount) as revenue
       from lineorder, dwdate
       where lo_orderdate = d_datekey
              and d_weeknuminyear = 6 and d_year = 1994
              and lo_discount between 5 and 7
```

```
            and lo_quantity between 36 and 40;
select sum(lo_revenue), d_year, p_brand1
      from lineorder, dwdate, part, supplier
      where lo_orderdate = d_datekey
            and lo_partkey = p_partkey
            and lo_suppkey = s_suppkey
            and p_category = 'MFGR#12
            and s_region = 'AMERICA'
      group by d_year, p_brand1
      order by d_year, p_brand1;
-- Consulta 4:
select sum(lo_revenue), d_year, p_brand1
      from lineorder, dwdate, part, supplier
      where lo_orderdate = d_datekey
            and lo_partkey = p_partkey
            and lo_suppkey = s_suppkey
            and p_category = 'MFGR#12'
            and s_region = 'AMERICA'
      group by d_year, p_brand1
      order by d_year, p_brand1;
-- Consulta 5:
select sum(lo_revenue), d_year, p_brand1
      from lineorder, dwdate, part, supplier
```

```
        where lo_orderdate = d_datekey
                and lo_partkey = p_partkey
                and lo_suppkey = s_suppkey
                and p_brand1 between 'MFGR#2221' and 'MFGR#2228'
                and s_region = 'ASIA'
        group by d_year, p_brand1
        order by d_year, p_brand1;
```

-- Consulta 6:

```
select sum(lo_revenue), d_year, p_brand1
        from lineorder, dwdate, part, supplier
        where lo_orderdate = d_datekey
                and lo_partkey = p_partkey
                and lo_suppkey = s_suppkey
                and p_brand1 = 'MFGR#2221'
                and s_region = 'EUROPE'
        group by d_year, p_brand1
        order by d_year, p_brand1;
```

-- Consulta 7:

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
        from customer, lineorder, supplier, dwdate
        where lo_custkey = c_custkey
                and lo_suppkey = s_suppkey
                and lo_orderdate = d_datekey
```

```
        and c_region = 'ASIA'
        and s_region = 'ASIA'
        and d_year >= 1992 and d_year <= 1997
    group by c_nation, s_nation, d_year
    order by d_year asc, revenue desc;
```

-- Consulta 8:

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
    from customer, lineorder, supplier, dwdate
    where lo_custkey = c_custkey
        and lo_suppkey = s_suppkey
        and lo_orderdate = d_datekey
        and c_nation = 'UNITED STATES'
        and s_nation = 'UNITED STATES'
        and d_year >= 1992 and d_year <= 1997
    group by c_city, s_city, d_year
    order by d_year asc, revenue desc;
```

-- Consulta 9:

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
    from customer, lineorder, supplier, dwdate
    where lo_custkey = c_custkey
        and lo_suppkey = s_suppkey
        and lo_orderdate = d_datekey
        and c_nation = 'UNITED KINGDOM'
```

```
            and (c_city='UNITED KI1' or c_city='UNITED KI5')
            and (s_city='UNITED KI1' or s_city='UNITED KI5')
            and s_nation = 'UNITED KINGDOM'
            and d_year >= 1992 and d_year <= 1997
      group by c_city, s_city, d_year
      order by d_year asc, revenue desc;
```

-- Consulta 10:

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
      from customer, lineorder, supplier, dwdate
      where lo_custkey = c_custkey
            and lo_suppkey = s_suppkey
            and lo_orderdate = d_datekey
            and c_nation = 'UNITED KINGDOM'
            and (c_city='UNITED KI1' or c_city='UNITED KI5')
            and (s_city='UNITED KI1' or s_city='UNITED KI5')
            and s_nation = 'UNITED KINGDOM'
            and d_yearmonth = 'Dec1997'
      group by c_city, s_city, d_year
      order by d_year asc, revenue desc;
```

-- Consulta 11:

```
select d_year, c_nation, sum(lo_revenue-lo_supplycost) as
profit1
      from dwdate, customer, supplier, part, lineorder
```

```
	where lo_custkey = c_custkey
		and lo_suppkey = s_suppkey
		and lo_partkey = p_partkey
		and lo_orderdate = d_datekey
		and c_region = 'AMERICA'
		and s_region = 'AMERICA'
		and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
	group by d_year, c_nation
	order by d_year, c_nation;
```

-- Consulta 12:

```
select d_year, s_nation, p_category, sum(lo_revenue-
lo_supplycost) as profit1
	from dwdate, customer, supplier, part, lineorder
	where lo_custkey = c_custkey
		and lo_suppkey = s_suppkey
		and lo_partkey = p_partkey
		and lo_orderdate = d_datekey
		and c_region = 'AMERICA'
		and s_region = 'AMERICA'
		and (d_year = 1997 or d_year = 1998)
		and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
	group by d_year, s_nation, p_category
	order by d_year, s_nation, p_category;
```

-- Consulta 13:

```
select d_year, s_city, p_brand1, sum(lo_revenue-lo_supplycost)
as profit1
      from dwdate, customer, supplier, part, lineorder
      where lo_custkey = c_custkey
            and lo_suppkey = s_suppkey
            and lo_partkey = p_partkey
            and lo_orderdate = d_datekey
            and c_region = 'AMERICA'
            and s_nation = 'UNITED STATES'
            and (d_year = 1997 or d_year = 1998)
            and p_category = 'MFGR#14'
      group by d_year, s_city, p_brand1
      order by d_year, s_city, p_brand1;
```